Universidad de Deusto

Análisis del desarrollo del Pensamiento Computacional con generalización mediante retos de programación visual

Tesis doctoral presentada por Andoni Eguíluz Morán
dentro del Programa de Doctorado de
Ingeniería para la Sociedad de la Información y Desarrollo Sostenible

Dirigida por Dra. Mari Luz Guenaga Gómez
y Dr. Pablo Garaizar Sagarmínaga

Universidad de Deusto

Análisis del desarrollo del Pensamiento Computacional con generalización mediante retos de programación visual

Tesis doctoral presentada por Andoni Eguíluz Morán
dentro del Programa de Doctorado de
Ingeniería para la Sociedad de la Información y Desarrollo Sostenible

Dirigida por Dra. Mari Luz Guenaga Gómez
y Dr. Pablo Garaizar Sagarmínaga

El doctorando

Los directores

Bilbao, mayo de 2020

Análisis del desarrollo del Pensamiento Computacional con generalización mediante retos de programación visual

Autor: Andoni Eguíluz Morán
Directora: Mari Luz Guenaga Gómez
Director: Pablo Garaizar Sagarmínaga

Puede encontrarse información acerca de esta tesis doctoral y los trabajos derivados de la misma en la siguiente dirección web:

http://link.eguiluz.net/tesis

Impreso en Bilbao
Primera edición, mayo de 2020

A Toñín y a María Luisa, a quienes debo el regalo más grande.

Lo hemos conseguido, papá.
Gracias por aceptarme y mirarme con amor desde ahí arriba.
Lo hemos conseguido, mami.
Gracias por aceptarme y tratarme con amor desde aquí abajo.

Gracias por tanto.

Resumen

La Informática y la serie de habilidades que la componen tienen una importancia creciente en las últimas décadas, que ha trascendido al mundo profesional y a la educación especializada. Su evolución ha dado lugar al desarrollo del concepto del Pensamiento Computacional, que unido a la prevista carencia de profesionales tecnológicos, ha llevado a una necesidad internacional de definir mejor ese nuevo tipo de conocimiento e incorporarlo en la educación obligatoria reglada, promovido por la explosión de materiales y herramientas que lo facilitan.

En esta investigación hemos diseñado y desarrollado una herramienta web específica, llamada Kodetu, orientada a aprendices jóvenes, que permite una experiencia autónoma de aprendizaje en forma de juego de niveles de dificultad creciente, resueltos en base a programación visual basada en bloques. Esta herramienta hace un registro detallado de las interacciones de los aprendices que nos permite estudiar con mucha precisión qué ocurre en los primeros minutos de contacto de una persona con los conceptos más elementales del Pensamiento Computacional.

Además de su uso libre a través de Internet, hemos definido grupos experimentales controlados de escolares que han trabajado con esta herramienta de forma autónoma, en sesiones de cincuenta minutos. Hemos recogido más de 6.000 sesiones de aprendices controlados y más de 3.000 de aprendices libres, con más de 5 millones de interacciones con las que hemos podido desarrollar un estudio basado en analítica de aprendizaje.

En paralelo, hemos diseñado y desarrollado una aplicación, llamada KodetuWin, que apoya en todas las fases del análisis: recoge todos los datos experimentales, permite hacer su limpieza y filtrado, calcula indicadores adicionales y genera tablas de datos y gráficos analíticos configurables. Las particularidades del aprendizaje del Pensamiento Computacional hacen posible simular la ejecución del código que el aprendiz modifica en cada interacción y que puedan generarse indicadores adicionales valiosos para el análisis del aprendizaje: tiempos, longitud de código, eficiencia de código, número de intentos, número de cambios hasta llegar a la solución, distancia al código exitoso, traza de código o código muerto. Estos indicadores aportan una visión más profunda de cómo ocurre el aprendizaje y proponemos que se utilicen en la construcción de la próxima generación de herramientas de aprendizaje del Pensamiento Computacional.

En el transcurso de nuestro estudio, hemos propuesto una manera innovadora de trabajar un aspecto particular de este aprendizaje, la competencia de generalización. Para ello, se realiza una segunda versión de la herramienta web que combina pares de retos que deben ser resueltos con el mismo código. En el estudio

comparativo de los aprendices de las dos versiones, descubrimos que la capacidad de enfrentarse a los niveles complejos es mejor cuando el aprendizaje ha ocurrido en la herramienta con generalización. Los aprendices encuentran más diversidad de soluciones, sus códigos se plantean de forma más generalizable y resuelven los retos en menos pasos. A cambio, el proceso es más lento y exigente, especialmente con los aprendices de edades menores o menor motivación. En base a nuestro experimento, proponemos el último ciclo de primaria (11 años) como el punto inicial en el que incorporar este tipo de trabajo. Proponemos la generalización como componente de influencia significativa en el desarrollo del Pensamiento Computacional y sugerimos que se tenga en cuenta en herramientas y contenidos escolares.

Estudiamos cuatro características personales (sexo, edad, conocimiento previo de programación y afinidad tecnológica) y aparecen claras correlaciones entre el desempeño y la edad. Se observa un mejor comportamiento de los chicos que de las chicas en los retos más mecánicos y una inversión de esta tendencia en los retos más abstractos, al incorporarse y combinarse las distintas estructuras. El comportamiento con respecto al conocimiento previo de programación es el esperado, mejor para los que lo tenían. En cuanto a la afinidad con la tecnología, el mejor grupo no es el esperable con la afinidad alta, sino el de media-alta, con lo que no es un buen predictor de comportamiento.

Las soluciones que los aprendices proponen para resolver los distintos niveles aportan mucha diversidad, a pesar de las evidentes limitaciones a las que les somete el sistema. En algunos niveles, la expresividad de código manifiesta largas colas que permiten analizar la originalidad de los aprendices y las relaciones entre esta creatividad y la eficiencia de código.

Entendemos que estas herramientas tienen un gran potencial para el aprendizaje del Pensamiento Computacional, pudiendo además servir al doble objetivo de formar y evaluar. Proponemos que se incorporen cuadros de mando asociados para que profesorado e investigadores tengan acceso a los resultados de la actividad de los aprendices, tanto a nivel individual como agregado.

Finalmente, a partir de nuestra experiencia hacemos una serie de recomendaciones para estos sistemas e indicamos aspectos clave a considerar para mejorar el aprendizaje: la incorporación de generalización, los límites de tiempo, los límites máximos de bloques, el registro fino de interacciones y los indicadores que pueden construirse y analizarse, la progresión de dificultad de los niveles y las estructuras de programación que se incorporan en cada uno, y la importancia de comprobar el comportamiento en esa progresión con experimentación en versiones preliminares.

Abstract

Computer science and its related disciplines have played an increasing importance role in recent decades and have transformed the professional world and specialized education. The evolution of computer science has led to the development of the concept of Computational Thinking. Together with the expected lack of technological professionals, an international need has emerged to define this new type of knowledge and incorporate it into compulsory education, promoted by contents and tools that simplify this new discipline.

In this research we designed and developed a specific web tool, called Kodetu. This tool is aimed at young learners and allows an autonomous learning experience in the form of a game of increasing difficulty levels that is solved based on visual block-based programming. This tool constructs a detailed record of learner interactions that allows us to study with great precision what happens in the first minutes of the experience with the most basic concepts of Computational Thinking.

In addition to its open Internet use, we defined controlled experimental groups of schoolchildren who have worked with this tool autonomously during fifty-minute sessions. We collected more than 6,000 sessions with controlled learners and more than 3,000 with free learners with more than five million interactions from which we were able to develop a study based on learning analytics.

At the same time, we designed and developed an application, called KodetuWin, which supports all phases of the analysis: it collects all experimental data, allows cleaning and filtering, calculates additional indicators and generates configurable data tables and analytical graphics. The singularity of Computational Thinking learning makes it possible to simulate execution of the code that the learner modifies during each interaction. Valuable additional indicators can be generated for learning analysis: times, code length, code efficiency, number of attempts, number of code changes, distance to successful code, trace of code and dead code. These indicators provide deeper insight into development of the learning process, and we suggest that these indicators be used in the construction of the next generation of Computational Thinking learning tools.

In the course of our study, we proposed an innovative way of working on a particular aspect of this learning, the generalization competence. For this method, a second version of the web tool was designed in which challenges in pairs that must be solved with the same code were combined. In the comparative study of learners from the two versions, we found that the ability to deal with complex levels is best when learning happens in tools with generalization. Learners find more diversity of code solutions, their codes are more generalizable and they solve challenges in fewer steps. In this exchange, the process is slower and more demanding, especially with younger or less motivated learners. Based on our experiments, we suggest that

the last cycle of primary school (11 years) should be the starting point for incorporating this type of work. We advise generalization as a component of significant influence on the development of Computational Thinking and suggest that it be taken into account in school tools and contents.

We studied four personal characteristics (sex, age, previous knowledge of programming, and technological affinity) and clear correlations between performance and age appeared. Better behavior was observed for boys than for girls in the challenges that were more mechanical. This trend was reversed in the more abstract challenges, as different structures were incorporated and combined. As expected, the behavior related to previous programming knowledge was better for those who had it. Regarding the affinity with technology, the best group was not the expected one that had high affinity, but rather medium-high affinity; thus, high affinity is not a good predictor of behavior.

The solutions that the apprentices bring to solve the different levels have a lot of diversity even with the obvious limitations forced by the system. At some levels, the expressiveness of the code manifests long queues that allows the originality of the learners and the relationships between this creativity and the efficiency of the code to be analyzed.

We understand that these tools have great potential for learning Computational Thinking, and they can also serve the dual objective of training and assessing. We suggest incorporating the associated dashboards for teachers and researchers to have access to the results of the activity of the apprentices, both individually and as an aggregate.

Finally, based on our experience, we make a number of recommendations for these systems and indicate key aspects to consider to improve learning: the incorporation of generalization, time limits, maximum block limits, fine logging of interactions and indicators that can be built and analyzed, difficulty in progression of the levels and the programming structures that are incorporated in each one, and the importance of checking the behavior in that progression with experimentation in preliminary versions.

Agradecimientos

Tendría que dedicar cien páginas si quisiera expresar todo el agradecimiento que tengo, debo y quiero. Haré un ejercicio de constricción para dejarlo solo en tres.

La tesis doctoral ha sido uno de los trabajos más difíciles de mi vida. Por la manera en la que funcionan mi naturaleza y aprendizajes, los trabajos de largo alcance no son los que mejor se me dan. Tampoco los trabajos que necesitan mucha memoria. Así que haber llegado hasta aquí y a esta edad es todo un logro… y en el primer y más importante lugar, os lo debo, Mari, Pablo. Os dije que no sabíais en lo que os metíais: a pesar de eso, habéis confiado, colaborado y apoyado este camino. Y con mayor mérito aún, lo hemos conseguido sin pelearnos, reforzando la admiración, amistad y alegría que nos une. Que dos de mis más maravillosos exalumnos hayáis sido los guías de este viaje me llena de orgullo y de agradecimiento. Qué preciosa profesión tenemos. Que la senda que nos queda sea al menos tan bella como la que ya hemos recorrido.

Enlazo con mis pasados directores de tesis y los que lo intentasteis en mis varios tropiezos: Javi O., Anselmo, JosuKa. De aquellos intentos beben estos logros. Gracias. Gracias, Asier P., Inés, por la confianza a pesar de los fracasos. Gracias a esas muchas personas que me habéis transmitido esa confianza que me costaba a veces encontrar: Javi G., Rebeca, Mari Jose, Arantza, Iñaki V., Asun, Charo, Begoña…

Gracias, Juanjo, mago de la estadística y compañero de color. Confío en que sea el primero de muchos trabajos compartidos y disfrutados.

Gracias, Miguel Ángel. Tu valiosa y desinteresada aportación a mi primer JCR desplegó las velas de esta singladura.

Este camino es un compendio de suertes, me empeño para que sean merecidas. El equipo de investigación del que formo parte es uno de esos regalos. No habría podido completar este trabajo sin vuestra impagable colaboración. Gracias, Ekaitz, por tantos ratos compartidos y porque sin ti Kodetu no sería ni Gen, ni Gami, ni nada (ánimo, que tú llegas aquí pronto). Gracias, Iratxe, Oihane, por todas esas sesiones que alimentan estos datos y por vuestra calidad humana (tranquilas, *que no me voy*). Gracias, Aitor, Alejandro, Olga, Ioanna.

Otro regalo para mí es la entidad en la que tengo la suerte de trabajar. Gracias, querida Uni. Sigo viviendo cada día, desde mis humildes capacidades, el "*sapientia melior auro*". El conocimiento es mejor que el oro. Cuánto lo necesita este mundo, y cuánto más desde la ética y el servicio como valores fundamentales.

Gracias a tantos compañeros y compañeras de Universidad (ya van para 30 años, quién lo diría). También mi admiración a todos los que ya habéis conseguido este logro: desde esta aventura os valoro bastante más. En esa larga lista de personas, esencial nuestro otro lema: "*el valor es la persona*". Gracias, Ander B., Iker J., David B., Alex, Diego L., Diego C., Susana R., Isabel F., Javi M., Enrique, Almudena…

Gracias a los 11.000 aprendices que habéis compartido aula conmigo. Tantos habéis mejorado con creces a vuestro profesor: qué mayor orgullo de un maestro. La Informática es ciencia y arte. Sigamos añadiendo siempre, siempre, el lado humano. Especial cariño y agradecimiento a esos muchos con los que supimos romper la barrera profesor/estudiante y crear fabulosos espacios horizontales, con tanta generosidad, creatividad, diversión y profundidad humana: el Grupo de Informática Social, el Aula Multimedia, la Semana ESIDE… seguiremos cocreando (en curso, TecnoÉtica y el Immersive Lab). Gracias, Borja, Gorka, Patxi, Miguel, Marta, David, Ibon, Susana V., Esther, Gari, Jorge G. (bis), Jorge L., Bingen, Jaime, Edu F., Marilou…

Gracias, amigas y amigos. Os atiendo mucho menos de lo que merecéis, confío estar a la altura cuando me necesitéis. Sois otro de mis regalos. El fiestón, cuando el coronavirus nos deje, va a ser de órdago. Un abrazo especial a mi querida cuadrilla: Beatriz, Pili, Karmelo, Ana, Pedro, Idoia, Iñaki, Nerea. Otro abrazo especial a mis queridas Adas: Cristina, Lorena, María, Estíbaliz; a mis queridos *tecnoéticos*: Peru, Galo, Olatz; a mis queridos *inmersivos*: Unai, Sergio, Álvaro. Gracias, Fernan, Jorge F., Roberto R., Vicente, Manu. No dejemos de celebrar la vida mientras nos dure la cuerda. Y que nos quiten lo *bailao*.

Gracias, Roberto F., por tu amistad y generosidad siempre disponible. No sé cómo habría llegado hasta aquí sin ti.

Gracias a mis maestras: Bea, Ana B., Nuria, Susana, Rosa. Mucho de lo bueno que tengo y soy lo he aprendido con y de vosotras. Soy mucho mejor hombre gracias a las mujeres con las que me he atrevido a compartir intimidad y camino.

Gracias, Ana, maestra y compañera. Qué regalo mutuo nos hizo la vida. Gracias por aceptarme y quererme tal y como soy, hasta en mis momentos surrealistas. Gracias por tanta risa, complicidad, confianza y alegría compartida.

Dejo para el final el mayor regalo y bendición. Josuan, Ander. Cuando la tesis fue alejándose de mí, no sabía que a cambio vosotros seríais testigos, ya de adultos, de la concreción de tantas de las ideas que hemos dialogado en nuestras deliciosas charlas. La importancia del trabajo, la creatividad, la pasión, el sacrificio, la apertura, la humildad. Con vosotros, *todo tiene mucho más sentido*. No olvidéis nunca a uno de nuestros filósofos de culto… el tío Ben: *"un gran poder conlleva una gran responsabilidad"* (recogida por el eterno Stan Lee, que la tomó prestada de Franklin D. Roosevelt). Sois grandes poderes. Además de un placer y un orgullo haberos apoyado hasta aquí, ahora es un gran placer y un orgullo veros volar con vuestros propios medios y recursos. Bea, hicimos mucho y bueno.

Toñín, María Luisa, este jalón es también vuestro. Gracias por vuestra humanidad, perseverancia, bondad y generosidad. Mi mayor fortuna fue nacer de vosotros. Gracias por todo.

A pesar de la tristeza y la desesperación, a pesar de los tiempos en que supuse que nunca lo lograría, a pesar de los años en que la mente parecía una sala de torturas y el cuerpo una prisión, a pesar de los años de dolor y alejamiento de mi verdadera naturaleza, ha sido una vida llena de bendiciones, una vida de riquezas indecibles, y no pudo haber sido de otra manera.

Y si todo terminara mañana, si cayera el telón, todo quedaría resumido en una sola palabra, tan solo quedaría una palabra, y esa palabra sería la misma que empezó todo esto. Esa palabra es Gratitud.

Gracias. Por todo: por la luz y la oscuridad, por las pérdidas y las ganancias, por el placer y el dolor, y por la inefable consciencia en donde todo surge y se desvanece como el canto de un pájaro.

Jeff Foster

Tabla de contenido

Índice de figuras

Índice de tablas

Índice de listados

I'm the kid who has this habit of dreaming
Sometimes gets me in trouble too
But the truth is, I could no more stop dreaming
Than I could make them all come true

Buddy Mondlock, Peter, Paul & Mary. "The Kid"

Capítulo

1

Introducción

Esta memoria de tesis doctoral recoge la investigación que hemos llevado a cabo en los últimos años a partir del desarrollo de una herramienta de aprendizaje autónomo de competencias básicas de programación, la incorporación explícita del proceso de generalización y la medición de su impacto en los resultados de los aprendices.

En este capítulo describimos la motivación y contexto que dan origen a esta investigación en la sección 1.1. Seguidamente, resumimos las hipótesis del estudio en la sección 1.2 y su metodología en la sección 1.3. Finalizamos con la sección 1.4 que muestra la organización general del documento.

1.1. Motivación y contexto

La Informática y la serie de habilidades que la componen tienen una importancia creciente en las últimas décadas. Su incorporación en titulaciones superiores y profesionales ha ido avanzando y propagándose a carreras no técnicas y a la formación continua. Esta presencia cada vez más ubicua y la prevista carencia de profesionales tecnológicos ha motivado la inclusión de forma generalizada de algunos de los aprendizajes que conlleva en los niveles obligatorios de educación. Esta necesidad, que ya se había explorado en los años 80 con los primeros lenguajes de programación textuales orientados a no profesionales, ha devenido en la incorporación de un nuevo concepto de Pensamiento Computacional (PC) y ha provocado la explosión de materiales y herramientas que lo intentan incorporar en niveles educativos básicos y medios.

El término PC, introducido por Jeannette Wing (2006), está conformado por una serie de habilidades cognitivas de resolución de problemas en las que la solución puede ser ejecutada por una computadora. Desgranaremos posteriormente ese concepto y veremos que en esa serie de habilidades destacan particularmente la abstracción y la generalización, muy relacionada con la anterior. En esta investigación, hemos explorado algunas maneras de incorporar este elemento de generalización, tan difícil de explicitar incluso en niveles educativos superiores, en herramientas de autoaprendizaje orientadas a aprendices jóvenes.

A este interés en el proceso de desarrollo del PC se une la experiencia de nuestro grupo de investigación (Deusto LearningLab[1]) en metodologías de analíticas de aprendizaje que permiten realizar análisis de experiencias educativas partiendo de recoger toda la información posible de la interacción de cada aprendiz con el sistema. Nos planteamos por ello la experimentación en PC utilizando herramientas digitales que registren al máximo el paso de los aprendices por el sistema, para descubrir en qué medida ese tipo de análisis puede mejorar el proceso de aprendizaje y también en qué medida puede ayudarnos a educadores y científicos a diseñar mejores herramientas para el mismo.

1.2. Hipótesis y preguntas de investigación

Tanto la ciencia en general como la ingeniería en particular tienen ciclos definidos y precisos que seguir. En ambos casos, los procesos pueden definirse de modo iterativo para realimentarse y enriquecerse. De esta manera, nuestra investigación comenzó con un objetivo que redefinimos en el camino y que recogemos en este documento.

El primer gran objetivo de nuestra investigación es conocer los detalles relacionados con el desarrollo del PC en aprendices noveles. Para ello, decidimos plantear una serie de retos de programación autónomos para registrar las primeras experiencias del PC de aprendices para determinar la influencia de las características personales, las diferentes soluciones que encuentran en los retos y los indicadores que podrían ayudar a plantear personalización y mejora de aprendizaje.

Dividimos este objetivo en diferentes preguntas de investigación, que codificamos de la siguiente manera:

- PI_K1. ¿Cuál es la influencia de las características personales (sexo, edad, afinidad tecnológica y conocimiento de programación previo) en las primeras experiencias del PC?
- PI_K2. ¿Cuántas soluciones diferentes encuentran los aprendices en sus primeros retos de programación y qué se puede concluir de estas diferencias?

[1] http://learninglab.deusto.es/

- PI_K3. ¿Qué variables podrían ayudar a plantear estrategias de personalización y adaptación de este tipo de herramientas de aprendizaje del PC?

Avanzando en el estudio, al darnos cuenta de la importancia de la generalización y de que las herramientas no la proponían explícitamente, valoramos la oportunidad de su incorporación a través de un rediseño de la herramienta y un segundo objetivo de investigación: plantear una serie de retos de programación que necesitaran generalizar (utilizando dos problemas que necesiten una misma solución) y registrar la experiencia de los aprendices para entender la influencia de la generalización y proponer una manera específica de hacerlo en herramientas de aprendizaje del PC.

Las preguntas de investigación en las que dividimos este nuevo objetivo son las siguientes:

- PI_G1. ¿Permite el análisis de las interacciones de un aprendiz en un sistema de estas características entender el desarrollo del PC?
- PI_G2. ¿Puede incluirse la competencia de generalización en los retos de programación de las plataformas lúdicas de aprendizaje autónomo de una manera eficaz?
- PI_G3. ¿Es la generalización un aspecto de influencia significativa en el desarrollo del PC? ¿Cómo impacta en el aprendizaje?
- PI_G4. ¿Cómo influyen las características personales (sexo, edad, afinidad tecnológica, y conocimiento de programación previo) en las primeras experiencias del PC y en su relación con la generalización?
- PI_G5. ¿Afecta la incorporación de la generalización a las soluciones encontradas por los aprendices?
- PI_G6. ¿Qué consideraciones deben tenerse en cuenta para diseñar este tipo de sistemas?

Esto determina la **hipótesis** de investigación que intentaremos validar: *El aprendizaje inicial del PC es un proceso sistematizable en el que la generalización tiene una influencia significativa. Puede modelarse a partir de las interacciones de cada aprendiz en plataformas lúdicas de aprendizaje autónomo, que permiten entender el desarrollo del PC a través de indicadores objetivos.*

Estas preguntas e hipótesis determinan las actividades que hemos llevado a cabo en esta investigación de cara a su validación.

1.3. Metodología de investigación

La metodología de investigación que hemos utilizado está representada conceptualmente en la figura 1.1. Resumimos a continuación el propósito de cada una de las fases:

Figura 1.1. Representación conceptual de la metodología de investigación.

1. Gestión del conocimiento

- Estudio de los antecedentes científicos existentes en los campos relacionados con nuestra investigación.
- Estudio de las herramientas tecnológicas relacionadas, así como las tecnologías de base para desarrollar nuestra propia herramienta.
- Elección de las tecnologías que utilizar.

2. Propuesta de hipótesis de investigación

En base al conocimiento adquirido:

- Formulación de las hipótesis y preguntas de investigación que dirigirán el proceso.
- Validación de que esas hipótesis reflejan la intención de nuestro estudio y de que no están previamente resueltas en la literatura.

3. Diseño e implementación de la plataforma

De acuerdo a las tecnologías definidas y a los objetivos de investigación:

- Diseño e implementación de la plataforma tecnológica que servirá de base para la experimentación con aprendices.
- Diseño e implementación de herramienta de análisis de datos.

4. Experimentación

- Diseño de la actividad experimental con aprendices seleccionados entre el público objetivo de nuestro estudio.
- Contacto con centros escolares y demás entidades, definición de agenda y equipamiento, organización de visitas y realización de sesiones.
- Comprobación de registro y validación de datos registrados.

5. Análisis experimental

- Tratamiento de los datos recogidos y filtrado.
- Observación de esos datos desde las distintas dimensiones de interés, con el uso de herramientas estadísticas y visualización y la propia herramienta de análisis de datos desarrollada ad-hoc.
- Obtención de resultados estadísticos y comprobación de validez.

En esta fase, en caso de detectar inexactitudes o no adecuación de la herramienta a los objetivos deseados, nos planteamos hacer nueva iteración de rediseño de herramienta y experimentación.

6. Difusión de los resultados

- Divulgación de los resultados obtenidos en congresos internacionales y revistas científicas, así como con la realización y posterior publicación de esta tesis doctoral.

Del mismo modo, en esta fase nos planteamos una iteración de revisión de objetivos de investigación y experimentación si es preciso.

1.4. Estructura del documento

Tras este primer capítulo introductorio, el capítulo 2 describe los estudios previos encontrados en las áreas científicas relacionadas con esta tesis. El capítulo 3 recoge la primera iteración de nuestro proceso de investigación, concretada en la herramienta que llamamos Kodetu y la experimentación con un colectivo de escolares y personas anónimas a través de Internet. El capítulo 4 inicia la segunda iteración de nuestra investigación, con el diseño de una nueva herramienta, KodetuGen. El capítulo 5 describe el rediseño de esta herramienta en KodetuGen2 y la experimentación realizada con ella en un colectivo de escolares en grupos

controlados. El capítulo 6 realiza una comparativa entre los resultados encontrados con las tres herramientas. Finalmente, el capítulo 7 hace un resumen de las aportaciones más relevantes de esta investigación y las líneas futuras que identificamos.

La creatividad te convertirá en un río.
Y te llenará de peces y criaturas.
Y la gente hará picnics junto a ti
y algunos nadarán en ti
y otros beberán de ti.
Y ese es el sentido de la vida.

Andréa Balt

The more I study, the more insatiable do I feel my genius for it to be.

Ada Byron

Capítulo

2

Estado del arte

Este capítulo revisa los conceptos y evidencias encontradas en la literatura científica en el contexto en el que se desarrolla nuestra investigación. Centraremos el propio término de pensamiento computacional y su diversidad de enfoques, para abordar después la problemática de su evaluación, especialmente en el entorno de las edades objetivo. Posteriormente, revisaremos las herramientas que se han utilizado para desarrollar ese aprendizaje, prestando particular atención a las de programación visual basada en bloques y revisando las plataformas existentes.

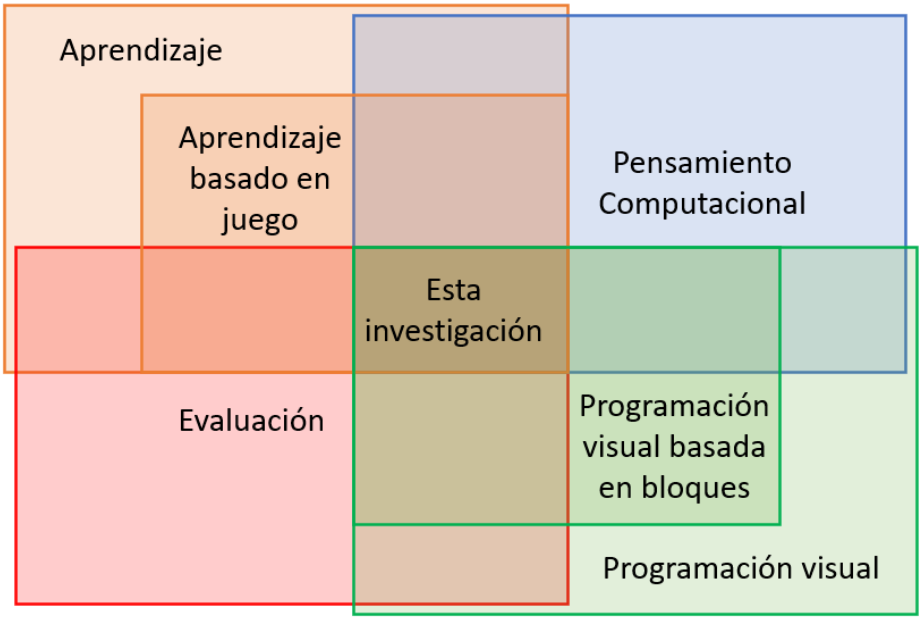


Figura 2.1. Representación de las áreas implicadas en esta investigación.

Mencionaremos también aspectos de aprendizaje basado en juego, para finalmente identificar los aspectos más relevantes en los que se centrará la investigación de capítulos posteriores. En la figura 2.1 se puede ver una representación conceptual de las áreas clave de este estudio.

2.1. Concepto de pensamiento computacional

Según Wing (2006), el Pensamiento Computacional (PC) está conformado por una serie de habilidades cognitivas de resolución de problemas cuya solución puede ser ejecutada por una computadora. En poco más de una década, desde que esta autora acuñara el concepto (introducido anteriormente por Papert, 1980), el movimiento ha sido muy intenso, tanto en la comunidad científica como en el mundo educativo, así como en las herramientas y contenidos disponibles (Buitrago Flórez y cols., 2017). Gracias a todo ese esfuerzo empiezan a verse cambios significativos en el terreno de la educación, aunque quedan muchos retos por resolver (Wing y Stanzione, 2016). Todavía es insuficiente la definición del concepto del PC (Denning, 2017) y poco consensuada su estructuración en el currículo educativo (Tedre y Denning, 2016), donde aparecen múltiples iniciativas de organizaciones nacionales e internacionales que se enfrentan a dificultades debido a la ya de por sí compleja estructura de la política educativa, en ocasiones distribuida por regiones (Wilson y cols., 2010).

El término PC es confuso en dos sentidos. En primer lugar, no es necesario un ordenador para desarrollar el PC. Muchas actividades permiten hacerlo con cartas, fichas, puzles o simplemente lápiz y papel, en lo que se ha denominado en castellano "informática desenchufada" (*Computer Science-CS unplugged*) (Bell y cols., 2015). El PC tiene que ver con una manera de pensar y de estructurar los problemas o procesos, que permiten plantear soluciones enunciadas de una forma sistemática, con un lenguaje formal cerrado. Una computadora es especialmente útil para implementar después esa solución. Merece la pena recordar que fue el mismo Alan Turing quien dijo que "la idea detrás de las computadoras digitales puede explicarse diciendo que esas máquinas se crearon para realizar cualquier serie de operaciones que podrían ser llevadas a cabo por un computador humano".

En segundo lugar, el PC parece ser una habilidad requerida solo para profesionales de la Ingeniería Informática, pero precisamente por su relación con los problemas y el enunciado de sus soluciones, y considerando lo ubicuo de las computadoras en nuestra sociedad digital, el PC es una habilidad fundamental para muchos sectores profesionales, y útil para cualquier profesional (Wing, 2006). Rushkoff (2010, p. 7) anticipa que en nuestra época digital las personas deben aprender no solo a usar programas sino cómo hacerlos (*program or be programmed*). Gardner y Davis (2013, p. 10) comentan que perseguir nuevas posibilidades gracias a nuestras *apps* nos hace activos mientras que solo usarlas nos hace dependientes (*app-enabling* vs. *app-depending*). En este sentido, se ha venido

acuñando el término inglés *computer literacy* (alfabetización informática) para referirse a la incorporación en el proceso educativo de maneras proactivas de construir tecnología, más allá de su simple uso (Rushkoff, 2010, p. 129). El trabajo de diSessa (2000) argumenta que la computación puede ser la base de una nueva forma de conocimiento con el potencial de permear entre materias, contextos y dominios.

En la última década, el PC ha ido ganando importancia dentro de la comunidad científica y educativa, proponiéndose como una habilidad fundamental que debe incluirse en el currículo educativo obligatorio. A nivel internacional hay una creciente tendencia a incorporarlo en educación primaria (Geldreich, Simon y Starke, 2019; Sabitzer y Demarle-Meusel, 2018; Bers y cols., 2014) y en secundaria (Djambong y cols., 2018; Zur-Bargury, Pârv y Lanzberg, 2013). Este movimiento ocurre en paralelo al impulso de las áreas STEM (*Science, Technology, Engineering, & Mathematics*) en la educación, y hay autores que otorgan al PC un papel relevante, si no central, entre las disciplinas STEM (Henderson y cols., 2007). Abundante literatura relaciona el PC con áreas educativas ya presentes en los currículos, como las matemáticas y las ciencias (Barbosa y Maltempi, 2019; Weintrop, Beheshti y cols., 2016), estudios sociales y lenguaje artístico (Barr y Stephenson, 2011), química y ciencias (Gautam, Bortz y Tatar, 2020), geometría (Echeverría y cols., 2019; Hutchins, 2018), música (Chong, 2019) o incluso ética (Seoane Pardo, 2018).

También en educación preuniversitaria y universitaria hay múltiples estudios que relacionan el PC con el mejor aprendizaje y comprensión de áreas no relacionadas directamente con la computación, particularmente dentro de las ciencias STEM, como probabilidad y estadística (Abrahamson, 2016), biología (Wilensky y Reisman, 2006), física (Perkins y cols., 2006) o física de partículas (Tinker y Xie, 2008). También en áreas como medicina (Saqr y Tedre, 2019) o comunicación con diseño de prendas electrónicas (Lui y cols., 2019).

Es tema actual de discusión cómo de importante es el PC, cómo tiene que ser de transversal en la educación obligatoria (Denning, 2017) y cómo debe integrarse en el currículo (Lye y Koh, 2014). También hay discusión sobre las bondades del PC: se afirma que el PC mejora habilidades cognitivas generales que se transferirán a otros dominios donde podrán mejorar la capacidad de resolver problemas más elaborados (Wing, 2016; Csizmadia y cols., 2015). Mark Guzdial (2015, p. 49-51) indica que no hay evidencias significativas que soporten esta afirmación, pero sí cree validadas muchas ventajas del PC, como dar soporte de base a otros aprendizajes. También se discute la diferencia entre el PC y la ciencia de la computación (Denning, 2009), y cómo están relacionados sin incluirse mutuamente.

No es el objeto de esta investigación profundizar en estas polémicas y lo cierto es que, motivado en gran medida por razones prácticas, el mundo educativo ha venido prestando al PC una considerable atención. Las autoridades educativas de

muchos países reconocen actualmente la importancia de incorporar la competencia del PC en sus distintas facetas (resolución de problemas, abstracción, simulación, descomposición, etc.), con apariciones ya significativas en el diseño curricular, como en Israel desde 2011 (Zur-Bargury, 2012). Poco después en Reino Unido se introdujo la informática de modo obligatorio en primaria y secundaria (Brown y cols., 2013, Royal Society 2012), y en muchos países europeos se viene observando el creciente reconocimiento de la importancia del PC (Balanskat y Engelhardt, 2015).

En respuesta al auge educativo, y también por la documentada carencia de profesionales de la computación en el presente y el futuro próximo, en la última década se han desarrollado muchas iniciativas a nivel local e internacional para promover el PC, con potencial de uso en educación primaria y secundaria, por ejemplo la hora de código[2] (*Hour of Code*), la semana de código[3] (*CodeWeek*) o el día del Scratch[4] (*Scratch Day*). Estos eventos globales apoyan los cambios del mundo educativo y suministran herramientas digitales de fácil acceso que permiten a los estudiantes introducirse en las habilidades básicas del PC.

De esta forma, los aprendices del PC pueden utilizar una gran variedad de herramientas para desarrollar esas habilidades, a menudo de forma autónoma, aunque también de forma supervisada por educadores. La mayoría de estas iniciativas se basan en plataformas digitales (mayoritariamente web) donde los programadores aprendices pueden desarrollar y mejorar sus habilidades del PC a través de juegos (Eguíluz, Garaizar y Guenaga, 2018).

Es importante definir con precisión el PC para poder estudiarlo e incorporarlo en procesos de aprendizaje. Para ello, primero lo diferenciamos de otros dos conceptos similares: programación de ordenadores y pensamiento algorítmico. La programación de ordenadores es el proceso con el que una persona es capaz de suministrar un conjunto de instrucciones que comunican lo más específicamente posible un procedimiento, método, práctica o tarea a una máquina. Estas instrucciones deben codificarse utilizando un lenguaje de programación específico (Robins, Rountree y Rountree, 2003). Esta labor tradicional de la enseñanza de la computación para futuros profesionales del área ha sido revisada en muchos estudios y tiene una dificultad reconocida, debida a su complejidad cognitiva y a las metodologías educativas (Jiménez-Toledo, Collazos y Revelo-Sánchez, 2019; Vihavainen, Airaksinen y Watson, 2014).

El pensamiento algorítmico es un concepto que está bastante cerca del PC y no toda la literatura los diferencia. Según Futschek (2006), el pensamiento algorítmico es una competencia cognitiva que permite obtener una solución a través de una serie de pasos, requiriendo para ello analizar problemas dados, especificar esos

[2] https://hourofcode.com/
[3] https://codeweek.eu/
[4] https://day.scratch.mit.edu/

problemas con precisión, dividirlos en partes si procede, encontrar las acciones básicas adecuadas y construir un algoritmo correcto con esas acciones.

Aunque no hay una única definición consensuada del PC, siguiendo las pautas de la principal introductora del término Jeannette M. Wing (2006, 2011), podemos verlo como *"los procesos de pensamiento involucrados en la formulación de problemas y sus soluciones, de modo que las soluciones se representen en un modo aceptable para un agente de proceso de información, humano, computadora o una combinación de ambos".* Aho (2011, 2012) lo simplificó definiéndolo como el conjunto de procesos mentales relacionados con la formulación de problemas cuyas *"soluciones puedan representarse con pasos y algoritmos computacionales"*.

Entendemos ampliamente aceptado que el PC incluye una serie de habilidades y prácticas de alto nivel que están en la base de la computación, que son aplicables a muchas áreas más allá de la informática, y que esas capacidades incluyen algunos conceptos reconocidos como clave (Selby y Woollard, 2013):

- abstracción: simplificar un sistema eliminando complejidad y detalles innecesarios. Es una manera de desentrañar sistemas complejos eligiendo los detalles que son relevantes, para poder ocultar los que no lo son. Esto puede hacerse de multitud de maneras, por ejemplo, cambiando la representación del sistema. Muchos autores y definiciones curriculares identifican la abstracción como la capacidad mental esencial del PC.
- generalización: resolver problemas nuevos basados en problemas previos resueltos. Relacionada con la abstracción, se refiere específicamente a cómo un mismo algoritmo puede generalizarse para resolver un conjunto de problemas que engloban el original y otros similares. También incluye la idea de reconocer y reutilizar partes comunes de una solución en otras, y lo que habitualmente se reconoce como patrones.
- descomposición: dividir una tarea (proceso, problema, sistema) en partes más pequeñas; o poder pensar en el conjunto en términos de sus partes. Las partes separadas pueden entenderse, resolverse y evaluarse por separado, lo que hace problemas grandes o complejos más fáciles de afrontar y resolver.
- pensamiento algorítmico: como acabamos de indicar, es una competencia cognitiva que permite obtener una solución a través de una serie de pasos.
- evaluación: asegurar que una solución es correcta, y que realiza su propósito (lo cual a menudo requiere compromisos entre la rapidez, el uso de recursos o la usabilidad).

De estas cinco habilidades, el pensamiento algorítmico incluido en la programación no solo es una de las actividades fundamentales que soporta las habilidades del PC, sino que también sirve como demostración objetiva de la competencia en el PC (Grover y Pea, 2013).

Hay otras habilidades y conceptos que suelen relacionarse con el PC con menos consenso (Grover y Pea, 2013, National Research Council, 2010). En general, están relacionados de alguna forma con los cinco mencionados. Indicamos los más significativos: gestión de datos/información (recolección, análisis, representación), sistematización, pensamiento lógico, pensamiento matemático, pensamiento procedimental, descomposición de problemas, diseño de sistemas, resolución de problemas, algoritmos y procedimientos, depuración, paralelización, automatización, modelado, simulación, recursividad y limitaciones de eficiencia y rendimiento (Barr y Stephenson, 2011). Fessakis y cols. (2018) hacen la revisión de una serie de materiales escolares significativos e identifican los conceptos de acuerdo a su presencia en las actividades. En orden de aparición, los más habituales son pensamiento algorítmico (66%), descomposición de problemas (50%), abstracción (47%), representación de datos (45%), razonamiento lógico (43%) y generalización (41%).

Kalelioglu, Gulbahar y Kukul (2016) realizan un metaanálisis de 125 estudios sobre el PC hasta 2014, donde identifican los principales conceptos relacionados con la definición del PC y los ordenan por apariciones. En orden decreciente: resolución de problemas, abstracción, computación, procesos, ciencia, datos, efectividad, algoritmo, conceptos, capacidades, herramientas y análisis. También recogen las competencias relacionadas con el PC y las ordenan de la siguiente forma: abstracción, pensamiento algorítmico, resolución de problemas, reconocimiento de patrones, pensamiento basado en diseño, conceptualización, descomposición, automatización, análisis, pruebas y depuración, generalización, razonamiento matemático, implementación de soluciones y modelado. Otros metaanálisis más recientes fundamentalmente confirman esta lista de competencias (Palts y Pedaste, 2020) y la reducen a ocho categorías de habilidades: formulación de problema, abstracción, reformulación de problema, descomposición, colección y análisis de datos, pensamiento algorítmico (incluyendo paralelización, iteración y automatización), generalización y pruebas y evaluación.

De los artículos revisados, obtenemos que las competencias más habituales referenciadas por los autores son, en este orden: pensamiento algorítmico, abstracción, descomposición, evaluación, generalización. Incluimos en el apéndice E la tabla de referencias manejadas a este respecto.

Hay otras maneras de desglosar el PC sin emplear una enumeración de competencias. A continuación, comentamos algunas de las más significativas en la literatura:

1) Relacionándolo con la ciencia y las matemáticas en una taxonomía de cuatro prácticas de aplicación del PC: datos, modelado y simulación, resolución de problemas de computación, y pensamiento de sistemas (Weintrop, Beheshti y cols., 2016).

2) Desde las habilidades de resolución de problemas que se aprenden con la programación: procesos y transformaciones, modelos y abstracciones, patrones y algoritmos, herramientas y recursos, inferencia y lógica, evaluaciones y mejoras (Gouws, Bradshaw y Wentworth, 2013).

3) Diferenciando tres dimensiones complementarias, según cómo se observe el comportamiento de los aprendices del PC: conceptos, prácticas y perspectivas (Lye y Koh, 2014; Brennan y Resnick, 2012). Los conceptos son los directamente utilizados por los programadores, como variables o sentencias. Las prácticas se refieren a prácticas de resolución de problemas que ocurren mientras se diseña el proceso como abstracción, depuración o remezcla. Las perspectivas tienen que ver con los aspectos personales de los aprendices como expresión, conexión y reflexión.

4) En un enfoque práctico, Lee y cols. (2011) proponen tres dimensiones útiles desde las que afrontar experiencias concretas del PC en jóvenes: abstracción, automatización y análisis.

Existen también definiciones operativas orientadas a la comunidad educativa. Una de las referencias más citadas se encuentra en la guía de recursos creada por las organizaciones estadounidenses ISTE (*International Society for Technology in Education*) y CSTA (*Computer Science Teachers Association*) (CSTA y ISTE, 2011). En esta guía, se define el PC como un proceso de resolución de problemas que incluye (al menos) las siguientes características: formulación de problemas de un modo que permita usar computadoras y otras herramientas electrónicas para ayudar a resolverlos; organización lógica y análisis de datos; representación de datos con abstracciones como modelos y simulaciones; automatización de soluciones con pensamiento algorítmico (como serie ordenada de pasos); identificación, análisis e implementación de posibles soluciones con el objetivo de obtener la combinación de pasos y recursos más eficaz y efectiva; generalización y transferencia de este proceso a una gran variedad de problemas. Por su parte, la organización británica CAS (*Computing at School*) (Csizmadia y cols., 2015) relaciona el PC con el razonamiento lógico e incluye las capacidades de pensamiento algorítmico, descomposición, generalización, patrones, abstracción, representación y evaluación. La ISTE actualiza (2016) las habilidades reconocibles en los estudiantes al respecto del PC: formular definiciones de problema preparadas para métodos tecnológicos y pensamiento algorítmico; recolectar datos o identificar conjuntos de datos relevantes con herramientas digitales para analizarlos; descomponer problemas en sus partes para extraer información clave y desarrollar métodos descriptivos de comprensión de sistemas complejos; y entender cómo funciona la automatización usando pensamiento algorítmico para elaborar una secuencia de pasos de solución.

Mannila y cols. (2014) realizan un estudio de percepción del PC entre el profesorado de secundaria del que identifican tres conjuntos de conceptos asociados: uno de datos (recolección, análisis, representación), un segundo de

abstracción (incluyendo descomposición de problemas y algoritmos) y un tercero de simulación, automatización y paralelismo.

Hemos revisado en esta sección la variedad de enfoques con los que se define y utiliza el PC. Para nuestra investigación, tomaremos en mayor consideración las competencias fundamentales (abstracción, generalización, descomposición, pensamiento algorítmico y evaluación), tendremos en cuenta que a través de la programación se puede objetivar el desarrollo de estas competencias y prestaremos en su momento especial interés a la de generalización.

2.2. Análisis y evaluación de aprendizaje del PC

Para comprobar la efectividad de cualquier currículo o herramienta utilizada en la educación es fundamental establecer medidas del aprendizaje. Existe una necesidad reconocida de evaluación de calidad y con instrumentos validados (Tew y Dorn, 2013). En esta sección, analizamos qué propuestas de evaluación sobre aprendizaje del PC se proponen en edades correspondientes a educación obligatoria.

2.2.1. Evaluación mediante cuestionarios y proceso manual

La evaluación del aprendizaje del PC puede realizarse con instrumentos convencionales, como cuestionarios de respuestas cerradas de opción múltiple. Lo habitual es que se evalúen actividades realizadas con herramientas específicas de programación visual, como Scratch[5] (Grover y Basu, 2017; Grover, Cooper y Pea, 2014), Alice[6] (Kelleher, 2006, p. 335-338) o Kodu[7] (Touretzky, Gardner-McCune y Aggarwal, 2017). También se utilizan cuestionarios mixtos con respuestas cerradas y abiertas para evaluar actividades con Scratch (Zur-Bargury, Pârv y Lanzberg, 2013), en este caso en pruebas a nivel nacional, en Israel. Varios autores como Lewis (2011) utilizan cuestionarios específicos basados en enunciados con ejemplos de código Scratch sobre los que se hacen preguntas con distintas combinaciones de planteamientos de comprensión: qué hace un código, qué resultado produce, qué falta en un código para que funcione, etc. Lewis lo utiliza buscando diferencias entre hacer programación individual y por pares.

La evaluación puede realizarse mediante análisis del código entregado, como hacen Werner y cols. (2012) utilizando Alice con estudiantes de secundaria: se pide a los aprendices (algunos individualmente y otros en pareja) que codifiquen partes de un código prediseñado para realizar tareas específicas, y se analizan posteriormente los métodos que han añadido o modificado, en función de las tareas propuestas. La evaluación puede ser así muy pormenorizada, pero es

[5] https://scratch.mit.edu/
[6] http://www.alice.org/
[7] https://www.kodugamelab.com/

difícilmente generalizable y automatizable. Este análisis se hace de modo manual también en otros estudios, como hacen Denner, Werner y Ortiz (2012) con juegos creados por niñas de secundaria usando Stagecast Creator, revisando el código de los participantes.

Franklin y cols. (2013) integran el PC en un campamento de verano de Scratch, programando por pares y evaluando la competencia demostrada sobre los proyectos entregados con rúbricas específicas de ejecución e implementación. En este proceso, los autores registran también las peticiones de ayuda a las que responden los instructores, diferenciadas en 4 niveles, desde validación (solo necesitan confirmación) hasta reformación (hay que volver a explicar el concepto).

Meerbaum-Salant, Armoni y Ben-Ari (2011) hacen análisis de código Scratch para buscar hábitos de programación inadecuados generados por el uso de la herramienta: una aproximación exclusivamente de abajo a arriba (*bottom-up*), tendencia a lo que califican como programación de grano extremadamente fino (información con mucho detalle), y uso no convencional e ineficiente de estructuras de programación. Los autores destacan que, pese a las ventajas de las herramientas abiertas, los conceptos base solo son bien aprendidos si se enseñan explícitamente mientras son aplicados en los proyectos.

Los mismos autores (2013) utilizan enfoques mixtos con algunos cuestionarios con preguntas cerradas sobre comportamiento de código, y tareas abiertas que proponen crear programas en Scratch que respondan a unos requisitos dados (con un especial énfasis en las taxonomías educativas como base para la evaluación). Scratch es una herramienta que se ha utilizado frecuentemente para introducir conceptos esenciales del PC. En la continuación de este estudio (Armoni, Meerbaum-Salant y Ben-Ari, 2015), los autores utilizan de nuevo este enfoque para estudiantes de último nivel de secundaria, esta vez con el aprendizaje de lenguajes de programación textual (Java o C#). Contrastan que los aprendices que previamente habían aprendido Scratch tienen mejor comportamiento en varios aspectos, más notoriamente en elementos abstractos que en conceptos básicos.

Seiter y Foreman (2013) proponen un modelo de progresión para relacionar prácticas de programación y edad con los proyectos registrados de los estudiantes (*Progression of Early Computational Thinking*, PECT). Para ello elaboran una rúbrica con evidencias en base a los bloques de Scratch utilizados, y otra más abstracta de diseño en base a los patrones de código Scratch que programa el aprendiz. La evaluación de la rúbrica se realiza de forma manual por expertos desde los códigos que desarrollan los estudiantes. En el análisis se observa cómo crece la complejidad del PC en los proyectos de Scratch de los estudiantes de una forma congruente con el nivel educativo, entre 1º y 6º de educación primaria.

Estos métodos solo cubren parcialmente las posibilidades de evaluación del PC, por lo que muchos investigadores han explorado nuevos procesos de evaluación especializados para esta habilidad, como el análisis del camino de aprendizaje de

los participantes en solución de proyectos abiertos (Fields y cols., 2016). Otros autores emplean la evaluación con análisis visual de portafolios en base a bloques (Brennan y Resnick, 2012), utilizando una herramienta llamada *Scrape* para diferenciar los tipos de bloques de Scratch utilizados y convertir los proyectos en una representación visual que rápidamente indica la cantidad de trabajo realizado y su tipología. Estos autores insisten en la necesidad de combinar múltiples medios de evaluación. No obstante, aún no se han desarrollado mediciones estandarizadas.

Usando su propio marco de trabajo, Lye y Koh (2014) realizan un metaanálisis de 27 estudios entre 2009 y 2013 sobre PC en todos los niveles educativos. Observan que en la mayoría se evalúan conceptos particulares del PC como bucles, o algoritmos particulares, mientras que en menos casos se observan las prácticas de pensamiento relacionadas con el PC, y solo en un par de casos se da importancia a las "perspectivas" del PC, entendidas como la capacidad de expresión que permite creación de narraciones personales. Lye y Koh identifican muchos puntos de mejora, entre ellos dar más importancia al proceso que al resultado, y hacer una captura más pormenorizada de esos procesos para poder analizarlos.

Vemos en esta serie de estudios la necesidad de criterios comunes para identificar y evaluar las habilidades incluidas en el PC. A este respecto, la existencia de herramientas digitalizables independientes de la plataforma, como el CTTest propuesto por Román-González, Pérez-González y Jiménez-Fernández (2017), puede ser un buen complemento para la evaluación en procesos de aprendizaje de medio plazo, tanto usando juegos como otras actividades. Los autores han diseñado y validado un cuestionario que mide el desarrollo del PC en el aprendiz compuesto por 28 preguntas de respuesta cerrada con cuatro opciones de respuesta sobre gráficos conceptuales que asemejan el funcionamiento de herramientas como Scratch o la Hora de Código pero son independientes de ellas. La edad objetivo es último ciclo de primaria y secundaria. Los conceptos del PC medidos son secuencias, bucles con contador, bucles condicionales, condicionales simples, condicionales compuestas y funciones sencillas.

Desde el punto de vista educativo, es fundamental que se desarrollen herramientas validadas como el CTTest. Aún hay pocas herramientas de PC que hayan pasado por un proceso de validación psicométrica, como también hacen Mühling, Ruf y Hubwieser (2015). Su cuestionario está diseñado para estudiantes de secundaria, y es independiente de la herramienta, aunque se basa en un lenguaje de programación textual genérico (pseudocódigo) con elementos similares a Karel el Robot (Pattis, 1981). Pretende medir las capacidades de los aprendices ejecutando un programa dado con estructuras de control de flujo, lo que ellos consideran el elemento central del PC para este grupo de edad: secuencias, selección (alternativas) y repetición; además de anidamiento de esas estructuras para crear programas más complejos.

También proponen una herramienta validada Weintrop y Wilensky (2015b) de lo que llaman "evaluación conmutativa", diseñada para estudiantes de bachillerato y segundo ciclo de secundaria. Este cuestionario mide la comprensión de algunos aspectos del PC y los autores lo utilizan para comparar el aprendizaje en entornos de programación visual basada en bloques con lenguajes de texto (JavaScript). El cuestionario tiene 28 preguntas y recoge los conceptos de condicionales, bucles, funciones simples, y funciones con parámetros. Lo plantean con cuestiones de comprensión sobre un código, de respuesta cerrada, con la originalidad de que la herramienta puede presentar ese código en su versión textual de pseudocódigo o en una versión visual de bloques. Las preguntas y sus respuestas se plantean en base a una serie de concepciones erróneas comunes en el aprendizaje de programación identificadas en la literatura (Sorva, 2012, p. 358-368).

El test FCS1 (Tew y Guzdial, 2011) se propone como una prueba de evaluación capaz de medir los conocimientos de Informática introductoria, diseñada para aplicarse de forma independiente a la pedagogía o al lenguaje de programación utilizado. En su estudio empírico mostraron que el pseudocódigo es un mecanismo apropiado para conseguir independencia del lenguaje de programación, y el FCS1 es un instrumento validado que desacopla los conceptos del PC del entorno o lenguaje utilizado para representarlos.

Basu y cols. (2020) usan un enfoque similar para desarrollar un cuestionario generalizable, que validan con más de 14.000 escolares de primaria (cursos 4º a 6º) de Hong Kong, desarrollando evaluación de conocimientos, competencias y habilidades de 4 áreas del PC: remezcla y reutilización, pensamiento algorítmico, abstracción y modularización y pruebas y depuración.

Buffum y cols. (2015) elaboran un protocolo de 7 pasos para diseñar, refinar y validar instrumentos de evaluación del PC. Ejemplifican este proceso en un contexto de formación de programación en secundaria usando aprendizaje basado en juego.

Otra serie de desarrollos proponen la evaluación en un plazo más extendido en el tiempo y como herramienta para la definición de políticas o de guía para los centros educativos o el profesorado. Dorling y Walker (2014) elaboran una rúbrica que diferencia una serie de áreas principales relacionadas con el PC (algoritmos, programación, datos, hardware, comunicación y tecnologías) y desarrolla niveles progresivos en cada una de ellas, basándose en experiencias de docentes y en el currículo nacional británico. Selby, Dorling y Woollard (2014) desarrollan una versión posterior que incorpora los conceptos del PC (abstracción, descomposición, pensamiento algorítmico, evaluación y generalización).

Una iniciativa europea realizada en esta dirección es el concurso internacional Bebras[8] (Cartelli, Dagiene y Futschek, 2010; Dagiene, 2005) que permite hacer un registro del aprendizaje del PC de jóvenes en edad escolar. Las preguntas del test

[8] https://www.bebras.org/

Bebras son independientes de la herramienta, son de varios tipos (pensamiento algorítmico, uso de sistemas digitales, estructuras, puzles, aspectos sociales) e incluyen algunas pruebas de programación con bloques, y también planteamientos mixtos: elegir instrucción que falta, elegir programa correcto de varias opciones, leer e interpretar un programa dado, etc. Este concurso se ha mencionado como un embrión para un futuro test PISA (*Programme for International Student Assessment*) en el campo del PC (Hubwieser y Mühling, 2014).

Existen además intentos de evaluación más allá de habilidades específicas, en el marco completo del PC. SRI Education (Bienkowski y cols., 2015) propone pruebas de evaluación a partir del diseño centrado en la evidencia (*Evidence-Centered Design*). Estas pruebas emplean modelos de cuestionarios con respuestas cerradas y abiertas para evaluar competencias clave del PC, en un contexto que denominan evaluación de principios del PC (PACT, *Principled Assessment of Computational Thinking*)[9]. A partir de este modelo, Grover (2017, 2015) combina en un curso con Scratch varias facetas de evaluación: rúbricas de evaluación sobre pequeños ejercicios individuales, un cuestionario final en base a código Scratch y evaluación manual de un proyecto abierto en parejas. La autora propone utilizar este tipo de evaluación combinada y lo denomina "sistemas de evaluación".

2.2.2. Evaluación mediante analítica de aprendizaje

Nos interesa conocer cómo pueden analizarse de forma automatizada las experiencias de aprendizaje del PC. Existe literatura que utiliza técnicas de analítica de aprendizaje (*Learning Analytics*, LA) y de minería de datos educativos (*Educational Data Mining,* EDM). Estos campos tienen muchas similitudes en su utilización y propósito, diferenciándose en 1) si el descubrimiento ocurre de forma automática o no, 2) el enfoque por componentes o sistémico, 3) los orígenes, 4) la adaptabilidad y 5) sus técnicas y métodos (Liñán y Pérez, 2015).

Se utilizan técnicas de EDM para modelar el comportamiento de estudiantes en actividades estructuradas de resolución de problemas (Baker, 2007). Particularmente en PC, se han definido varios modelos para entender cómo los aprendices desarrollan esta habilidad. Werner, McDowell y Denner (2013a, 2013b) siguieron una aproximación analítica para describir cómo los estudiantes de secundaria programan con Alice, introduciendo la importancia que pueden tener los registros de ejecución de bajo nivel, tanto con una tarea abierta como con otra de planteamiento cerrado. Se describe la importancia del análisis de la estructura de los datos registrados y cómo se relacionan con acciones de alto nivel de los participantes (teniendo que construir un analizador sintáctico y crear métricas específicas para convertir de forma automática los registros en acciones de alto nivel). Los autores indican el gran potencial que tiene el análisis de registros para

[9] https://pact.sri.com/

entender cómo los jóvenes aprenden con la tecnología, especialmente en entornos de aprendizaje no estructurado como entornos de programación para aprendices.

Alice es uno de los primeros entornos de aprendizaje del PC en el que se utilizan registros para investigación. Kelleher (2006) plantea su tesis doctoral construyendo un sistema de registro en Alice para recoger las actividades fundamentales de alto nivel realizadas (programación, edición de escena, y reproducción).

De forma similar, Maloney y cols. (2008) realizan un estudio con 536 proyectos de Scratch evaluando, entre otras cosas, los registros semanales de los ficheros de resumen de proyecto, que incluyen el número y tipo de bloques usados, así como los personajes (*sprites*) y sonidos incorporados en cada proyecto. Este estudio observa el incremento en el uso de los conceptos de programación (bucles, variables, lógica, aleatorios) a lo largo de un año.

Gujberova y Kalas (2013) utilizan el concurso internacional Bebras, mencionado anteriormente, para hacer un registro del comportamiento de escolares del primer ciclo de primaria (8-10 años). El registro que realizan los autores es de grano fino, diferente dependiendo de cada tipo de prueba.

El proceso de LA puede realizarse también sobre grandes volúmenes de datos. Dasgupta y cols. (2016) utilizan el registro de una de las plataformas más globales (Scratch) para analizar la evolución de más de un millón de usuarios que hacen remezcla (*remix*). Muestran con ello que la remezcla influye positivamente en el aprendizaje de las habilidades incluidas en el PC, comprobando cómo esos usuarios introducen los tipos de instrucciones en sus proyectos nuevos si los han utilizado antes en proyectos remezclados. Proponen una diferenciación de seis ejes relacionados con el PC en función de los bloques de Scratch utilizados: bucles, paralelismo, eventos, condicionales, operadores y datos.

Hay muchas investigaciones que utilizan lenguajes de programación convencionales. Hemos identificado algunas donde se usan técnicas de EDM y LA basadas en registros de grano fino de la interacción de los usuarios, para extraer patrones de sus comportamientos. Spacco y cols. (2015) utilizan CloudCoder[10], una herramienta de código abierto que permite proponer ejercicios a los estudiantes en lenguajes fuente (C/C++, Java, Python, o Ruby) y que almacena no solo las respuestas finales de los participantes, sino el registro de grano fino de las ediciones (inserción y borrado de caracteres) que van ocurriendo en cada uno de los ejercicios que responde el usuario.

Piech y cols. (2012) identifican técnicas de aprendizaje automático (*machine learning*) concretadas en un modelo de Markov para describir cómo los estudiantes encuentran soluciones, utilizando 200.000 registros de código de estudiantes

[10] http://cloudcoder.org

noveles de programación universitarios, partiendo de variantes del lenguaje de programación Java sobre un contexto definido de dos ejercicios simplificados. Los patrones que encuentran predicen mejor el desempeño de los aprendices en sus exámenes que las evaluaciones recibidas en las prácticas de programación entregadas: es decir, la manera en la que se crea un programa es mejor predictor del aprendizaje que el propio programa resultado. Según los autores, este tipo de análisis podría ser utilizado en estudiantes universitarios de programación, registrando los estados progresivos de sus códigos en un lenguaje de programación convencional como Java.

Vihavainen, Luukkainen y Ihantola (2014) realizaron un curso en línea masivo y abierto (MOOC) de introducción a la programación con más de mil estudiantes, usando el entorno integrado de desarrollo (IDE) NetBeans[11] para que los estudiantes fueran realizando los ejercicios del curso. Con la instalación del *plugin* Test My Code[12] los estudiantes pueden no solo probar el código sino realizar entregas, y además el *plugin* hace registro de datos de grano fino (capturas -*snapshots*-, registro de estado del sistema en cada momento) y pulsaciones de teclas. Los autores defienden la conveniencia de hacer registros de grano fino donde se encuentra información relevante que habría pasado desapercibida analizando solo las entregas finales.

En la misma línea, Blikstein y cols. (2014) realizan también capturas de código fuente Java y Karel el Robot y miden su evolución entre estados consecutivos, en primer lugar en número de líneas y caracteres de código insertados, borrados y modificados; después con métricas de similitud entre programas de diferentes aprendices usando palabras clave, llamadas a la API, y cambios en el árbol sintáctico del código. Encuentran que, al examinar el proceso de la programación (la evolución del código de cada aprendiz) en lugar del resultado final, se encuentran datos contraintuitivos sobre el comportamiento de los estudiantes de programación. Añaden que en ese análisis pueden encontrarse patrones con mejor predicción de competencia que los exámenes convencionales.

Observamos que un buen número de estudios utilizan juegos como soporte de aprendizaje para desarrollar el PC. Los juegos proponen un reto cuantificable, medible y observable, oportuno para realizar una evaluación. De hecho, en algunas plataformas que mencionaremos posteriormente ya se están considerando indicadores cercanos a la evaluación tales como el tamaño de código (número de bloques) en code.org[13], o la puntuación asociada al desempeño en tiempo o nivel en the Foos[14], entre otros.

[11] https://netbeans.org/
[12] http://testmycode.github.io/
[13] https://code.org/
[14] https://codespark.com/

La evaluación no es trivial en soluciones de programación. Es deseable que incluya indicadores clave en habilidades del PC, como eficiencia de programa (número de pasos de ejecución), calidad programática de la solución encontrada, o evaluación del comportamiento del usuario en el proceso (número de errores, número de cambios de código, camino seguido hasta la solución correcta, etc.). Del mismo modo que un docente de programación con un lenguaje de programación textual (C, Python o Java, por ejemplo) no tiene una rúbrica sencilla para evaluar a sus estudiantes, también puede ser complejo para un docente de primaria o secundaria, que no es necesariamente un experto en computación ni está acostumbrado a identificar los aspectos claves de calidad del software, más allá del funcionamiento correcto de la solución planteada. Es muy relevante en este sentido el marco que proponen Grover y cols. (2017a) para combinar métodos basados en hipótesis con analítica de datos para procesar registros de datos de herramientas de PC, lo que denominan analítica combinada (*blended analytics*).

Vemos que el análisis de registros suele ocurrir a posteriori, pero también hay algunas experiencias que recuperan y procesan esa información en directo, de manera que puedan utilizarse como parte síncrona del proceso formativo. Koh y cols. (2014b) desarrollan una evaluación formativa en tiempo real para el progreso de estudiantes y grupos (REACT: *Real Time Evaluation and Assessment of Computational Thinking*). A diferencia de otros métodos, este está orientado al profesorado para la supervisión en tiempo real de la actividad de sus estudiantes, y se realiza mediante el análisis del código AgentSheets[15] (o AgentCubes) de los proyectos de los aprendices. Está basado en un modelo de evaluación desarrollado por Koh y cols. (2010), validado por el mismo Koh y cols. (2014a). Este modelo, CTPA (*Computational Thinking Pattern Analysis*), desglosa nueve competencias relacionadas con el PC que se pueden desarrollar en la elaboración de juegos digitales, y permite visualizar un gráfico de desempeño combinado en esas competencias, que corresponden a patrones del PC relacionados con juegos (Basawapatna, Koh y Repenning, 2010): control de usuario, generación (de objetos de juego), absorción (eliminación de objetos), colisión, transporte (un objeto lleva a otro), empuje (un objeto empuja a otro), tiro (un objeto tira de otro), difusión (un objeto difunde elementos como energía a objetos cercanos) y escalado (algoritmos de búsqueda).

En general, las herramientas específicas existentes para el mundo escolar carecen aún de las posibilidades, automáticas o manuales, para que los educadores puedan realizar una monitorización progresiva de la experiencia de los aprendices en entornos abiertos. Se necesitan nuevos mecanismos de análisis y evaluación para poder validar en qué medida los estudiantes van más allá de resolver un problema, y estudiar cómo lo resuelven y cómo progresan. Probablemente, subsistemas del

[15] https://agentsheets.com/

tipo propuesto por Moreno-León (2015) en Dr. Scratch[16] se irán incorporando para facilitar indicadores que enriquezcan el proceso tanto para aprendices como para sus supervisores. La herramienta, gratuita y de código abierto, parte de cualquier proyecto público de Scratch y devuelve una serie de valores tabulados de 0 a 3: paralelismo, pensamiento lógico, control de flujo, interactividad con el usuario, representación de la información, abstracción y sincronización. Muestra además una valoración de puntuación general e identifica una serie de malos hábitos (programas duplicados, nombres de objetos inadecuados, código muerto, etc.). Dr. Scratch se basa en dos proyectos previos, Scrape (Wolz, Hallberg y Taylor, 2011) y Hairball (Boe y cols., 2013). En la misma línea, Alves y cols. (2020) proponen una rúbrica automatizable, CodeMaster, para evaluar proyectos realizados en App Inventor.

Fields y cols. (2016) parten de capturas progresivas, cada dos minutos, del fichero JSON que describe el programa Scratch de cada uno de los participantes, que convierten en indicadores cuantitativos que reflejan aspectos del PC en el desarrollo de proyectos abiertos, como inicialización, eventos y paralelismo. También describen la elaboración de un proyecto de fuente abierto (*FUN! Tool*)[17] para realizar mediciones automatizadas en entornos como Scratch.

La creciente relevancia del LA y el EDM ha hecho aparecer repositorios abiertos para el almacenamiento y acceso a datos de registro para investigación educativa, como el PSLC Datashop[18] (Koedinger y cols., 2010).

Puede consultarse en el apéndice F un listado completo de las referencias consultadas de evaluación de PC para nuestra investigación.

2.2.3. Otras técnicas de evaluación y aprendizaje

Hemos revisado un buen número de aproximaciones que permiten enfocar el aprendizaje del PC y evaluarlo. Aunque no serán objeto de nuestra investigación, queremos también reseñar que hay múltiples estudios sobre aprendizaje y evaluación del PC que no enfocan el PC de forma directa, sino combinada con otros aprendizajes.

Lu y Fletcher (2009) proponen utilizar el concepto del lenguaje del PC (*Computational Thinking Language, CTL*) para incorporarlo en asignaturas de otras áreas. Marshall (2011), por su parte, intenta evaluar cómo los estudiantes son capaces de reconocer patrones del PC en clips de vídeo. Miller y cols. (2014) integran ejercicios del PC con ejercicios de pensamiento creativo para analizar su impacto en calificaciones en los primeros cursos universitarios de computación.

[16] http://www.drscratch.org/
[17] https://github.com/ActiveLearningLab
[18] https://pslcdatashop.web.cmu.edu/

Hershkovitz y cols. (2019) integran retos de PC con actividades de creatividad para relacionar la creatividad computacional con el pensamiento creativo.

Hay toda una serie de publicaciones que no incluimos en este capítulo que evalúan el PC en relación con la robótica u otros elementos físicos, lo cual es especialmente interesante en aprendices preescolares o de primeros ciclos de primaria, y en una rama del PC que, por el hecho de incluir objetos manipulables y con comportamiento físico, se viene denominando "tangible" (Bers, 2017; Kazakoff y Bers, 2012). Yu y Roque (2018) hacen un estudio de 34 de estas herramientas en el que identifican cinco categorías: 1) físicos con electrónica, 2) físicos sin electrónica, 3) virtuales, 4) híbridos con bloques virtuales, y 5) híbridos con bloques tangibles.

2.3. Herramientas para aprendizaje de programación

En la primera parte de la revisión hemos visto un número considerable de menciones a una serie de plataformas como Scratch (Resnick y cols., 2009), Alice (Cooper, Dann y Pausch, 2000), code.org (Kalelioğlu, 2015) o AgentSheets (Repenning, 2000). Estas herramientas han ido apareciendo en paralelo a la necesidad de incorporar el PC en el mundo educativo, muchas de ellas públicas y gratuitas, y aportan no solo un recurso valioso para el aprendizaje y la evaluación del PC, sino una gran oportunidad para la investigación, como hemos visto en la utilización de LA.

Debido a esa importancia, dedicamos esta sección a revisar las herramientas tecnológicas aplicables al campo de nuestro estudio, empezando con un repaso histórico desde sus primeras apariciones en el mundo educativo.

2.3.1. Primeros intentos de programación en las escuelas

En los primeros tiempos de la Informática se vio que el diseño de un lenguaje de programación específico podría ayudar a los estudiantes a entender la computación, por lo que ya a finales de los 60 empezaron los primeros diseños de lenguajes de programación más accesibles (Mendelsohn, Green y Brna, 1990).

Seymour Papert no es solo el padre del concepto del PC, también es uno de los diseñadores de Logo, considerado como el primer lenguaje de programación de la historia con propósito educativo (Feurzeig y cols., 1970). Logo es un lenguaje de programación de propósito general que se empezó a experimentar asociado a la enseñanza de matemáticas en estudiantes de primaria y que pasó a tener un importante uso internacional en los 80 en entornos educativos. El lenguaje Logo introdujo una serie de características que serán muy importantes en fases posteriores, como un avatar móvil visible en pantalla o los comandos de movimiento "egocéntrico", que se expresan en primera persona sobre ese avatar (adelante, gira a la derecha, etc.). Este avatar se definió originalmente como una tortuga, que dibujaba en pantalla en función de movimientos vectoriales dirigidos

desde el código (Solomon y Papert, 1976). También incorporó primitivas y sintaxis de lenguaje muy accesibles para usuarios sin formación específica previa.

Aunque el diseño original pretendía enseñar conceptos matemáticos, sus creadores descubrieron rápidamente el potencial educativo adicional de Logo: "la presencia de la computación puede contribuir a procesos mentales no solo instrumentalmente, sino también en maneras esenciales y conceptuales, influyendo en cómo las personas piensan, incluso cuando no tienen contacto físico con un ordenador" (Papert, 1980, p. 4).

El siguiente lenguaje significativo diseñado para aprendices fue BASIC, que es un juego de palabras entre "básico" y el acrónimo *Beginner's All-purpose Symbolic Instruction Code*. Sus características fundamentales para su uso en el contexto educativo son un conjunto de instrucciones limitado, eliminación de toda la sintaxis innecesaria, y mucha cercanía entre la edición y la ejecución de programas (Reed y Palumbo, 1988). Tras él, se desarrollaron una larga lista de opciones de lenguajes de texto: lenguajes para paradigmas de programación complicados de incorporar en aprendices novatos como la orientación a objetos con el lenguaje Blue (Kölling y Rosenberg, 1996) o minilenguajes que no pretenden ser lenguajes de programación de propósito general (Brusilovsky y cols., 1997) como Karel el Robot (Pattis, 1981).

Una segunda línea de entornos de aprendizaje del PC tiene relación con otra de las pioneras fundamentales de la informática educativa, Radia Perlman. Entre 1974 y 1976 desarrolló el primer sistema de programación tangible, llamado TORTIS (Perlman, 1976), orientado y probado con niños y niñas preescolares (3-5 años). Su objetivo específico era "superar el obstáculo del teclado y hacer que muchas de las ventajas proporcionadas por el aprendizaje de lenguajes informáticos completos sean accesibles para niños de tan solo tres o cuatro años".

Este sistema TORTIS se inspira en el conjunto de comandos de tortuga de logo, y permite programar el comportamiento de un robot tortuga físico (con desplazamiento, luz, bocina y lápiz) en función de una serie de acciones básicas (movimiento y activación de los elementos del robot), que se introducen a través de dos terminales con botones que permiten secuenciar las acciones básicas, combinarlas en acciones más complejas, así como memorizarlas y repetirlas.

Los sistemas tangibles de programación permiten a los niños/as expresar programas a través del movimiento de objetos, en lugar de escribir comandos o manipular iconos digitales. En algunos casos, el resultado de programar esos sistemas se visualiza en una pantalla de ordenador, como es el caso de AlgoBlock (Suzuki y Kato, 1993). En otros, ocurre a la inversa: el programa se introduce en un software de ordenador o dispositivo móvil, y se ejecuta en un robot o dispositivo físico. Esta variante tiene multitud de ejemplos en nuestros días, como varias de las iniciativas de los juguetes Lego, como trenes (McNerney, 1999) o ladrillos de construcción para primaria (Pinto-Llorente y cols., 2016).

Otro enfoque observa los usos que se dan a la programación y el PC para el aprendizaje. Edith K. Ackermann (2012) estudia cómo los niños y niñas usan la programación para explorar la relación entre ellos y los dispositivos digitales. Identifica tres roles diferentes: 1) dar instrucciones para que un dispositivo haga lo indicado, 2) enseñar a que un dispositivo tenga autonomía para comportarse por sí mismo y 3) remezclar y modular comportamientos partiendo de programas existentes y modificando algunas de sus partes.

Con cualquiera de estos enfoques, una de las problemáticas fundamentales que se ha encontrado al analizar cómo los aprendices se enfrentan a las habilidades de programación es la complejidad de la sintaxis. La investigación ha descubierto que esas reglas establecidas sobre un lenguaje reducido e inicialmente desconocido, con sus arbitrarias normas de puntuación y de formato, es una seria barrera de entrada para los programadores novatos (Denny y cols., 2011) y que las características de la sintaxis influyen directamente en la facilidad del aprendizaje (Stefik y Siebert, 2013).

Hay múltiples maneras de abordar la simplificación de la sintaxis, como narrativas, diagramas de flujo o construcciones especializadas (Powers, Gross y cols., 2006). Vamos a centrarnos en la programación visual basada en bloques, que en la literatura y en la práctica es la herramienta más significativa y utilizada, especialmente cuando nos referimos a niños y jóvenes.

2.3.2. Programación visual basada en bloques

Muchas de las herramientas que han ido apareciendo en este capítulo (Scratch, Alice, Code.org) utilizan un enfoque de programación basado en bloques visuales (*block-based programming*) en lugar de codificación basada en texto. Vemos ejemplos del aspecto visual de esos bloques de algunas de las herramientas más conocidas en la figura 2.2.

El fundamento conceptual de estas herramientas se basa en descubrimientos antiguos en la Informática, y particularmente en los llamados editores estructurados. Donzeau-Gouge y cols. (1980) parten de la unidad atómica de composición que es un nodo del árbol sintáctico abstracto (AST, *Abstract Syntax Tree*) de un programa, en lugar de un elemento más pequeño como puede ser un carácter o un símbolo de puntuación en un lenguaje de programación de texto. Al representar los bloques constructivos sobre esos nodos, el sistema puede asegurar que los bloques se localicen solo en lugares válidos del AST general del programa, evitando de esta forma por completo los errores de sintaxis (además de los léxicos). Es paradójico leer en el informe de Donzeau-Gouge que la posibilidad de editar programas directamente en la pantalla no sería una tarea fácil, y que incluso no sería deseable.

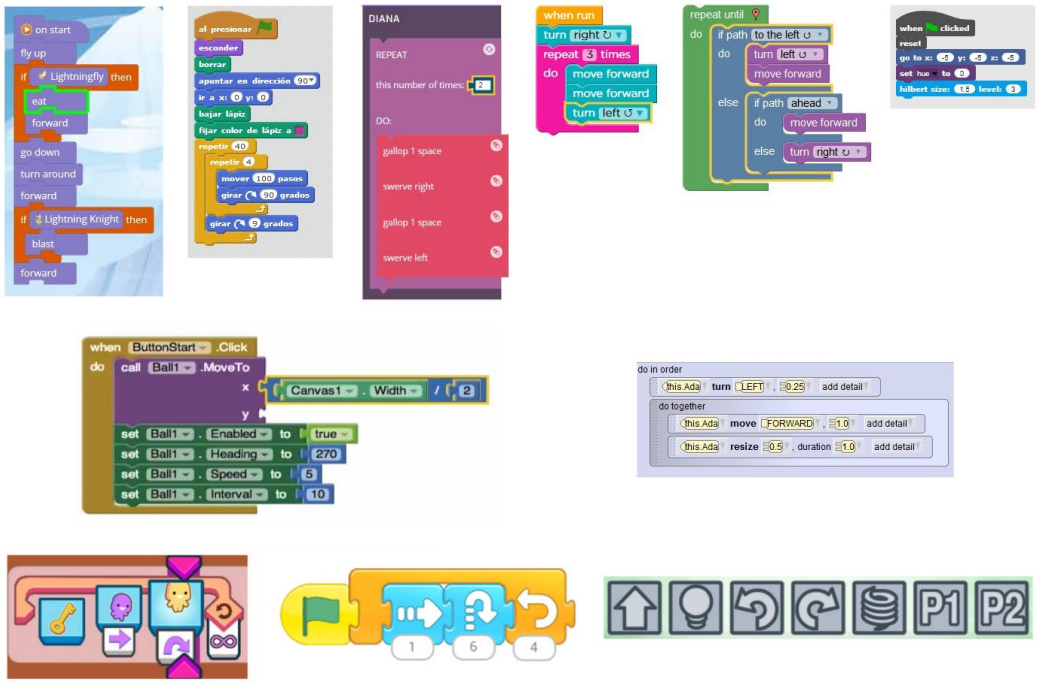


Figura 2.2. Ejemplos de bloques de algunas herramientas. Arriba Tinker, Scratch, Made with Code, code.org, Blockly, BeetleBlocks. Hilera central: AppInventor, Alice. Abajo: The Foos, ScratchJr, Lightbot.

Pocas décadas después, ya hay una serie importante de lenguajes de programación visuales basados en bloques que han ido desarrollando y mejorando este concepto. Entre los primeros intentos significativos encontramos BridgeTalk (Bonar y Liffick, 1987) y LogoBlocks (Begel, 1996). Después apareció Alice, muy utilizado en informática preuniversitaria y universitaria en sus primeros cursos. A partir de ahí, una larga lista que revisaremos posteriormente.

Estas herramientas basan la interfaz de usuario en bloques visuales que se mueven y colocan como un juego de montaje, usando habitualmente la metáfora visual de los puzles, con sus piezas y formas de encaje. Estos bloques funcionan como abstracción de los componentes de programación: sentencias, datos, estructuras de control, procedimientos, etc. En consecuencia, limitan considerablemente el conocimiento previo que hay que tener para programar, y refuerzan la estructura del programa, eliminando la distracción de la sintaxis y enfocando en la lógica que existe en la actividad que se quiere plantear. Siguiendo la metáfora, los aprendices pueden componer programas funcionales usando solo el ratón para arrastrar y combinar piezas, recibiendo retroalimentación (*feedback*) visual inmediato de cuándo las construcciones son válidas o no lo son (Bau, Gray y cols., 2017).

Usualmente, cada bloque suministra pistas visuales (color, forma o tamaño) que representan la lógica combinatoria entre piezas, y utiliza etiquetas de lenguaje natural (“while”, “if”, “forward”, etc.) para representar su funcionalidad. Además se suelen usar otras pistas visuales, como color para diferenciar uso conceptual (por

ejemplo, diferenciando por colores los tipos de bloques como en Scratch) o forma visual englobadora como metáfora de anidamiento/ámbito.

En la figura 2.3 vemos un ejemplo de la pantalla de edición de Scratch, en la que se aprecian los espacios más habituales en este tipo de herramientas (etiquetadas como A, B, C, D). Las herramientas suelen suministrar un espacio origen (A) en el que aparecen todos los bloques disponibles, agrupados de distintas maneras dependiendo de la herramienta. Desde ese espacio, el aprendiz puede localizar y arrastrar cada bloque a otro espacio diferente (B) de código (o de trabajo o de construcción) en el que los bloques pueden agruparse componiendo programas que pueden después ejecutarse. En muchas herramientas esta ejecución produce un *feedback* visual en el espacio de código, en lo que sería la metáfora equivalente a la depuración en programación convencional. Para completar este planteamiento existe un espacio (C) de escenario (o ejecución) donde se visualiza el resultado de la ejecución, sustituyendo a la consola de programación convencional. En Scratch, como en muchas de las herramientas más elaboradas, existen zonas adicionales (D), como es el caso del espacio de *sprites* o el de configuración.

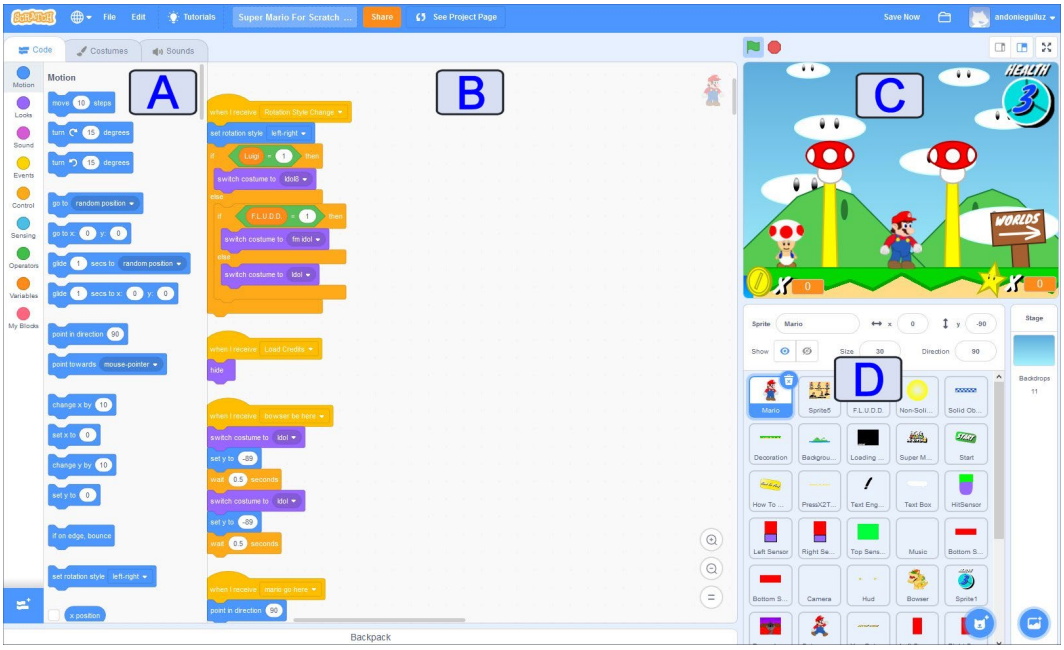


Figura 2.3. Pantalla ejemplo de Scratch. A: Espacio origen. B: Espacio de código. C: Escenario. D: Espacio de *sprites* y configuración.

Aunque la distribución cambia en cada herramienta, los tres primeros espacios nombrados son muy habituales. En una aplicación más sencilla como Kodable[19] (figura 2.4) los podemos identificar igualmente.

[19] https://www.kodable.com/

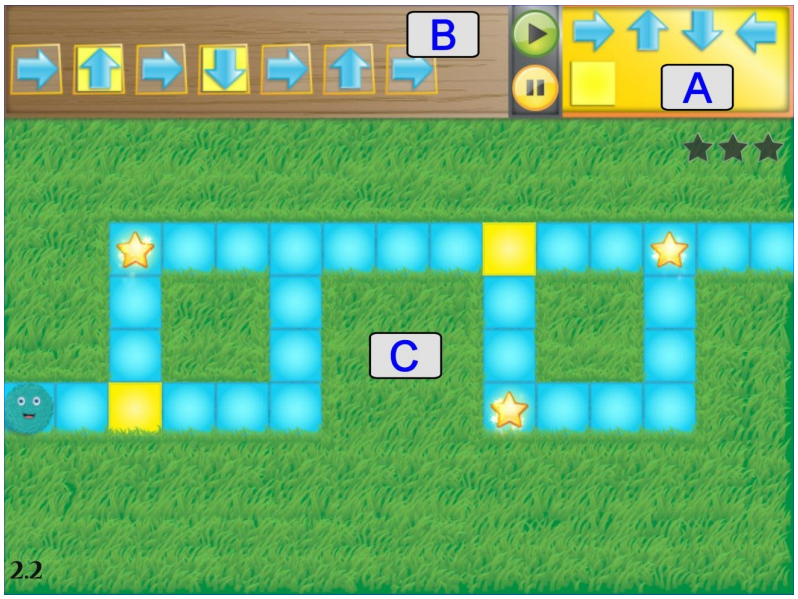


Figura 2.4. Aspecto de interfaz de Kodable. A: Espacio origen. B: Espacio de código. C: Escenario.

Varias de las metáforas visuales que se usan son muy relevantes. Como hemos mencionado, las piezas pueden tener salientes o huecos conectores con formas geométricas que ayudan a la colocación: no todo bloque encaja con cualquier otro. Es el caso de Scratch (figura 2.5). Este es uno de los elementos fundamentales de la conversión de una sintaxis léxica, muy proclive a errores y distante de la humana, a una visual más fácil de entender y que prácticamente evita los errores de sintaxis (Bau, Gray y cols., 2017). Este concepto ha sido muy explorado en las herramientas desarrolladas en el mercado y en la literatura (Lerner, Foster y Griswold, 2015), e incluso existen propuestas en las que se pueden configurar piezas con salientes y entrantes personalizados definiendo lógicas de encaje en función de su contexto (Roque, 2007).

Figura 2.5. Bloques de código con detalle de encajes en Scratch.

Otro aspecto significativo es que la secuencialidad se representa por contigüidad visual, bien sea vertical (como en el caso de Scratch) u horizontal (como en Kodable

o Scratch Jr.). De este modo en la ejecución de código los bloques se ejecutan de una forma natural para los aprendices: de arriba abajo o de izquierda a derecha.

Los bloques de la zona origen están a la vista para poder ser arrastrados con el ratón (o de forma táctil) hasta el espacio de código. En entornos complejos como Scratch o Alice, la cantidad de bloques disponibles hacen imposible esta composición y suelen entonces agruparse en pestañas, jerarquías o paletas.

Es también reseñable la importancia de la nomenclatura que se utiliza en estas herramientas. Las orientadas a las edades más tempranas intentan evitar al máximo el texto y usan metáforas visuales. Las que incorporan texto reducen al máximo la jerga y aprovechan conceptos comunes y conocidos. Los nombres que se eligen tienen impacto en la facilidad de aprendizaje, según Stefik y Seibert (2013).

Otro elemento clave en la facilidad de aprendizaje de estas herramientas son las abstracciones que los diseñadores de las herramientas eligen para que los usuarios comprendan rápidamente el comportamiento que tendrá un programa cuando una de las construcciones se utilice. Por ejemplo, los diseñadores de Alice se enfrentaron al reto de desarrollar abstracciones intuitivas para controlar animaciones 3D. Para ello trabajaron con usuarios en el proceso de diseño, y sustituyeron aspectos complejos como matrices de transformación por conceptos más intuitivos como movimientos relativos a objetos (Conway y cols., 2000).

Debido a la proliferación de este tipo de herramientas en la última década y a su continua evolución, se pueden identificar muchos otros pequeños aspectos que algunas de las herramientas consideran y que mejoran la comprensión de su funcionamiento. Nombramos algunos (Bau, Gray y cols., 2017):

- Contexto de parámetros (figura 2.6, izquierda arriba). Los bloques que necesitan información (por ejemplo, la nota a reproducir y su duración) completan con palabras significativas esa información componiendo frases fáciles de entender (*play note X for Y beats*). Esto es especialmente útil cuando hay varios parámetros en el mismo bloque.

Figura 2.6. Ejemplos de características de herramientas. (izda.) Contexto de parámetros y editores especializados. (cent.) Representación de estado. (dcha.) Minimización de elecciones relevantes.

- Editores especializados. Cuando uno de los datos que se necesita es un número en un rango, se puede mostrar un deslizador en lugar de un cuadro de texto. Cuando el número representa a una frecuencia de una nota musical, puede aparecer un teclado con la nota (figura 2.6 izquierda). Cuando el dato corresponde a una lista cerrada de opciones, se muestra un desplegable, etc.
- Representación de estado. En la ejecución de código, se puede mostrar el estado dinámico según se van ejecutando los bloques. Por ejemplo, con efectos visuales (transparencia, brillo, borde) sobre el bloque en curso o vistas del valor contenido en las variables (figura 2.6 centro).
- Minimización de las elecciones relevantes. Los lenguajes de programación textuales añaden muchos elementos sintácticos que distraen del significado principal. Estas herramientas permiten solo observar o modificar los elementos clave para ese significado. Por ejemplo, una construcción repetitiva con contador donde en ciertos niveles lo único relevante es el número de repeticiones. En otro contexto se pueden añadir la variable, los valores extremos, la condición de finalización y el incremento. Por ejemplo, en la figura 2.6 pueden verse a la derecha tres variantes del mismo bloque repetitivo *for*. El superior es el que tendría cualquier lenguaje de programación textual, con 17 elementos diferentes. Los dos inferiores son versiones que code.org muestra en una de sus actividades, reduciendo el número de elementos relevantes dependiendo del nivel del aprendiz. En general, poder prestar atención visual solo a los aspectos significativos disminuye la carga cognitiva de la sentencia lógica representada.
- Los bloques directamente manipulables permiten experimentar pequeños fragmentos de código sin necesidad de conectarlos con el programa principal, algo mucho más difícil en lenguajes textuales. Es lo que destacan Meerbaum-Salant, Armoni y Ben-Ari como programación *bottom-up* (2011). En los entornos que además soportan modificación en directo (*liveness*), esos bloques pueden ejecutarse simplemente con clics de ratón, o los valores de las variables pueden actualizarse mientras el sistema se está ejecutando, llevando uno de los beneficios de los lenguajes interpretados a las herramientas de programación visual (Maloney y Smith, 1995).
- Depuración avanzada. Hay conceptos de depuración avanzada como la ejecución reversible (Zelkowitz, 1973) que se conocen ya desde los 70 y que se incorporan en algunos de los entornos de programación orientados a bloques, aunque por el momento son más frecuentes los entornos con lenguajes textuales convencionales como UUhistle (Sorva y Sirkiä, 2011) u Online Python Tutor (Guo, 2013).

- Un último elemento reseñable generalizado en algunos de estos entornos es el acceso a código ajeno, compartido a través de repositorios o comunidades, junto a la posibilidad de lo que se viene denominando remezclar: modificar un proyecto de otro usuario para hacer una versión modificada, o simplemente para aprender. Literatura muy abundante refleja las ventajas de la remezcla, tanto en lenguajes convencionales (Brandt y cols., 2010; Dorn y Guzdial, 2006) como en estas herramientas de programación visual basada en bloques. Es el caso de Scratch (Dasgupta y cols., 2016), donde tanto se pueden importar proyectos completos de otros usuarios, como utilizar la llamada "mochila" para guardar partes de código y recursos que pueden compartirse entre proyectos.

Descripción de herramientas de PC de programación visual

No es objetivo de esta investigación discutir las ventajas de incorporar el PC en la educación, sino analizar las herramientas, sus posibles usos y sus limitaciones, y proponer mejoras en las actualmente utilizadas. Por ello, realizaremos una revisión de algunas de las plataformas existentes y analizaremos sus características de cara a obtener información útil para nuestra propuesta.

Para delimitar la revisión, constatamos que hay muchas características comunes entre la mayoría de las herramientas actuales orientadas al desarrollo de habilidades del PC como code.org, Alice o Scratch. En primer lugar, son abiertas y gratuitas para su uso general. En segundo lugar, se orientan fundamentalmente a estudiantes de primaria y secundaria. En tercer lugar, hay una característica de juego más o menos explícita que facilita el uso de esos aprendices y utiliza los beneficios comentados de los juegos en la educación. Es también importante cómo todas estas herramientas buscan evitar que los programadores novatos se enfrenten a la complejidad de la codificación de computadoras basada en texto a través de la programación visual, con las ventajas ya comentadas. Para esta revisión nos centraremos en las herramientas más habituales que son las que utilizan programación visual basada en bloques. Será también el tipo de herramienta que desarrollaremos y utilizaremos para nuestra investigación.

Para determinar el conjunto de plataformas existentes con estas características encontramos una serie de artículos científicos que describen o enumeran algunas de ellas (Jiménez-Toledo, Collazos y Revelo-Sánchez, 2019; Yu y Roque, 2018; García-Peñalvo y cols., 2016b; Laporte y Zaman, 2016; Harteveld y cols., 2014; Malliarakis, Satratzemi y Xinogalos, 2014; Vahldick, Mendes y Marcelino, 2014; Gibson y Bell, 2013; Malliarakis, Satratzemi y Xinogalos, 2013; Li y Watson, 2011; Sandoval-Reyes, Galicia-Galicia y Gutierrez-Sanchez, 2011; Daly, 2009). Nos planteamos la necesidad de realizar un análisis objetivo incluyendo el mayor número posible de herramientas, y revisando sus posibilidades desde distintas

perspectivas de una forma consistente, lo cual concretamos en un capítulo de libro (Eguíluz, Garaizar y Guenaga, 2018) que incluimos completo como anexo C en este documento (traducido a castellano).

En esa revisión realizamos una comparativa entre un buen número de las plataformas *online* para PC gratuitas (o parcialmente) disponibles, limitándonos a las que no requieren hardware adicional, basadas en programación visual y con alguna característica de juego incorporada. Diferenciamos el análisis en varias dimensiones: 1) Pedagógicas, características que pueden afectar al proceso, tiempo o nivel de aprendizaje; 2) Punto de vista de juego, la diversión y la implicación generados; 3) PC, revisando los conceptos implicados en el PC que cubre cada plataforma y en qué extensión; y 4) Otros aspectos de diseño, como el registro o la personalización y adaptación a las características personales de los aprendices de programación, sus habilidades y conocimientos.

Resumimos a continuación los elementos más significativos de este estudio. Se encontraron y estudiaron 24 plataformas con las características indicadas, mencionándose además un buen número de sistemas que no entraban en ellas.

Se encuentran dos grandes grupos de plataformas: los juegos de retos de PC (46,2% de las analizadas) y los lenguajes de programación visual (42,3%). Los primeros (como code.org) son juegos organizados en niveles predefinidos, de dificultad e incorporación de conceptos del PC progresivos, en general con tiempos cortos de uso (el juego se acaba cuando se resuelve el último nivel), entre pocas horas y pocas semanas. La evaluación (en modo de puntuación o de imposibilidad de avanzar) está implícita en esta categoría.

Los segundos (como Scratch o Alice) son entornos abiertos en los que los aprendices pueden desarrollar sus propios programas, sin retos prefijados, como ocurre con un lenguaje de programación convencional. En ese sentido, estos entornos no incluyen la dinámica de juego de forma explícita, pero la funcionalidad y los ejemplos disponibles hacen que los elementos que más habitualmente se desarrollan en ellos sean juegos, animaciones o historias interactivas. Esos sistemas son generalmente ajenos a la evaluación y secuenciación que se haga con ellos.

Un tercer y muy poco habitual tipo de plataformas (7,7%) son las que permiten no solo resolver retos sino también crearlos, incluyendo de esta manera dos opciones diferenciadas de abordar el PC: resolver un reto programando y crear un reto para que haya que programar. Son MakeWorld[20] (Guenaga y cols., 2017) y Robozzle[21].

Una última categoría ejemplarizada por Minecraft[22] corresponde a videojuegos que incorporan el PC en sus mecánicas.

[20] https://makeworld.eu/
[21] http://www.robozzle.com/
[22] https://www.minecraft.net/

Las opciones sociales de estas plataformas (marcar preferidas, compartir, seguir, etc.) apenas son soportadas por los juegos de retos, mientras que sí son muy habituales en los entornos abiertos. Hay diversidad de herramientas que cubren todas las edades, enfocándose las de retos de programación a edades menores y prefiriéndose los entornos abiertos a partir de secundaria. También hay herramientas para cubrir diferentes dedicaciones, desde unas pocas horas (lo habitual en los sistemas de retos) hasta cursos completos (normalmente con enfoques de aprendizaje basado en proyectos, con entornos abiertos).

Se realiza una medición de la implicación (*engagement*) de los sistemas que relaciona los más utilizados con el mayor esfuerzo de diseño por conseguir que sean atractivos al incorporar elementos diversos que favorecen la diversión. Es el caso de Minecraft, code.org, Tynker[23], Scratch, Hopscotch[24], Alice y Snap![25].

Al respecto de las características específicas del PC incluidas, se observa que se incluyen en mayor medida cuanta mayor complejidad y edad objetivo de la herramienta. Los juegos de retos incluyen en su mayoría las estructuras básicas, aunque es reseñable que ninguno aborda de forma explícita el concepto de generalización. Es destacable que un concepto tan abstracto como la recursividad sí se incorpora en varias herramientas, en forma de llamadas al mismo código para realizar repetición (virtualmente infinita) en sistemas donde no se incluyen bucles: es el caso de Cargo-Bot[26], LightBot[27], MakeWorld y Robozzle. Observando solo los entornos abiertos, se encuentra que incluyen grandes juegos de bloques básicos con los que programar con más expresividad (todos excepto Hopscotch y ScratchJr[28] tienen más de 100 tipos de bloques). También todos soportan eventos, lo cual constata la importancia de la programación orientada a eventos en la informática actual, siendo también de un entendimiento muy natural para los aprendices. La mayoría permiten programación multihilo, aunque sea de una forma transparente al aprendiz, que utiliza el concepto sin probablemente comprenderlo. Hay una serie de características que solo se incorporan en algunos de los entornos abiertos: paso de mensajes, orientación a objetos, 3D o conexión con sistemas físicos. Solo una herramienta (Alice) incorpora la construcción de secuencia paralela.

Entre otros aspectos de diseño, destaca la ausencia de adaptabilidad al comportamiento del usuario (los sistemas no cambian en función de cómo el usuario lo haga de bien o de mal, o de su edad). La opción vertical de código es la más habitual (69,2%) y en una cuarta parte se usa la opción horizontal (23,1%), siendo la preferencia bloques para el interfaz vertical en edades superiores, e iconos para el horizontal en edades inferiores. Hay dos herramientas que no encajan en

[23] https://www.tynker.com
[24] https://www.gethopscotch.com/
[25] https://snap.berkeley.edu/
[26] https://itunes.apple.com/es/app/cargo-bot/id519690804
[27] https://lightbot.com/
[28] https://www.scratchjr.org/

estos esquemas: Kodu que propone un interfaz de grafo donde las opciones se van realizando por niveles y en cada nivel se muestran con iconos las disponibles, en forma de círculo; y Bee-Bot[29], que no tiene espacio de código explícito: es cada aprendiz quien tiene que recordar de memoria la secuencia de programa que carga en la abeja protagonista (del mismo modo que en el dispositivo físico Bee-Bot).

Algunas herramientas incluyen funcionalidad específica para formadores, permitiendo gestionar grupos de estudiantes y suministrando información en forma de cuadro de mando visual para consultar el comportamiento de grupo y aprendices particulares. Destacan en esta faceta code.org, Kodable y Tynker.

Se recogen como limitaciones el uso excesivo de primitivas de movimiento y rotación en dos dimensiones (algo que ocurre desde el Logo original) por encima de otras opciones visuales o auditivas; la carencia de acceso a datos de uso para analítica de aprendizaje e investigación educativa en general; la falta de adaptabilidad al usuario; y la falta de mecanismos de evaluación y monitorización más detallados y adaptados al PC.

Los entornos abiertos como Scratch pueden ser aplicados para aprendizaje autónomo, y es un uso que se promueve (Maloney, Peppler y cols., 2008), incentivado por la propia comunidad de desarrolladores y la fácil remezcla de otros proyectos. Sin embargo, este enfoque tiene importantes carencias de aprendizaje si no se combina con una formación específica y orientada a los conceptos de programación bien aplicada (Meerbaum-Salant, Armoni y Ben-Ari, 2011).

Estos entornos siguen uno de los principios principales de entornos de programación para jóvenes que ha persistido desde los tiempos de Logo: "suelo bajo, techo alto" (*low floor, high ceiling*) (Grover y Pea, 2013). Implica que es fácil para un principiante cruzar el primer umbral de crear programas funcionales (suelo bajo), a la vez que la herramienta es suficientemente expresiva y potente para programas más avanzados (techo alto). Repenning, Webb y Ioannidou (2010) hablan de las características que deben tener estas herramientas para provocar "impacto sistémico": umbral de entrada bajo, techo alto, despliegue fluido, posibilidad de transferencia, soporte de igualdad y sostenibilidad, y universalidad de uso.

En la tabla 2.1 puede verse una lista de las herramientas analizadas para este estudio, con sus citas iniciales o más significativas en la literatura y su dirección web (activa en marzo de 2020).

[29] https://itunes.apple.com/es/app/bee-bot/id500131639

Herramienta	Referencia	Web
Alice	(Cooper, Dann y Pausch, 2000)	http://www.alice.org/
App Inventor	(Wolber y cols., 2011)	http://appinventor.mit.edu/
Bee-Bot	(Mannila y cols., 2014)	https://itunes.apple.com/es/app/bee-bot/id500131639?mt=8
Beetle Blocks	(Koschitz y Rosenbaum, 2012)	http://beetleblocks.com/
Blockly Games	(Grover, Bienkowski y cols., 2016)	https://blockly-games.appspot.com/
Cargo-Bot	(Gibson y Bell, 2013)	https://itunes.apple.com/es/app/cargo-bot/id519690804
Code.org	(Kalelioğlu, 2015)	https://code.org/
Daisy the Dinosaur	(Laporte y Zaman, 2016)	https://apps.apple.com/us/app/daisy-the-dinosaur/id490514278
DeltaTick	(Wilkerson-Jerde y Wilensky, 2010)	http://ccl.northwestern.edu/rp/deltatick/
EvoBuild	(Wagh y Wilensky, 2012)	http://ccl.northwestern.edu/~awagh/evobuild.html
Kodable	(Mannila y cols., 2014)	https://www.kodable.com/
Kodetu	(Eguiluz y cols., 2020)	http://kodetu.org/
Kodu Game Lab	(Touretzky, Gardner-McCune y Aggarwal, 2017)	http://www.kodugamelab.com/
Hopscotch	(García-Peñalvo, Rees y cols., 2016a)	https://www.gethopscotch.com/
Lightbot	(Gouws, Bradshaw y Wentworth, 2013)	https://lightbot.com/
LogoBlocks	(Begel, 1996)	https://www.media.mit.edu/projects/logo-blocks/overview/
Made with Code	(Mannila y cols., 2014)	https://www.madewithcode.com/
MakeWorld	(Guenaga y cols., 2017)	https://makeworld.eu/
Pocket Code	(Slany, 2014)	https://share.catrob.at/app/
RoboBuilder	(Weintrop y Wilensky, 2012)	http://ccl.northwestern.edu/roboBuilder/
Robozzle	(Gibson y Bell, 2013)	http://www.robozzle.com/
Scratch	(Resnick y cols., 2009)	https://scratch.mit.edu/
ScratchJr	(M. U. Bers, 2017)	https://www.scratchjr.org/
Snap!	(Weintrop y Wilensky, 2015a)	http://snap.berkeley.edu/
SpriteBox	(Eguíluz, Garaizar y Guenaga, 2018)	http://spritebox.com/
Stagecast Creator	(Denner, Werner y Ortiz, 2012)	http://acypher.com/creator/
StarLogo TNG	(Begel y Klopfer, 2007)	https://education.mit.edu/project/starlogo-tng/
The Foos	(García-Peñalvo, Rees y cols., 2016a)	http://thefoos.com
Tynker	(García-Peñalvo, Rees y cols., 2016a)	https://www.tynker.com
Waterbear	(Vasek, 2012)	http://waterbearlang.com

Tabla 2.1. Listado de herramientas de programación visual basada en bloques.

2.3.3. ¿Lenguajes de bloques o lenguajes de texto?

Hemos comentado cómo la programación visual basada en bloques ha llegado a su difusión masiva los últimos años de la mano del auge del PC. Disminuye significativamente la complejidad con respecto a programar en un lenguaje de programación basado en texto (como Python, C o Java), reduciendo la dificultad de aprender una sintaxis particular y permitiendo limitar o eliminar por completo los errores de sintaxis tan habituales en la actividad de programación de software. Weintrop y Wilensy (2015a) demuestran que las interfaces de usuario de programación visual basadas en bloques son más fáciles de usar por sus abstracciones (piezas de código en lugar de sentencias) e interacción (arrastre de ratón) más naturales. Estos autores utilizan Snap!, uno de los primeros entornos de desarrollo que permiten una combinación entre bloques visuales y lenguaje textual (JavaScript). En sus conclusiones destacan que los bloques son más fáciles de leer, sus formas son útiles, son más fáciles de componer, y evitan tener que memorizar. También tienen inconvenientes: la programación es menos potente, más lenta y se siente menos auténtica.

Bau y cols. (2017) hablan de un momento reseñable para los lenguajes basados en bloques, comentando también cómo las herramientas visuales disminuyen la barrera de entrada y mejoran el aprendizaje. Además de insistir en las ventajas del propio concepto de los bloques, destacan cómo el aprendizaje mejora con elementos asociados como entornos en línea; abstracciones de alto nivel; el concepto de paleta organizado por función, que favorece la exploración y el descubrimiento; editores especializados de argumentos; información de estado visible; o ejemplos sencillos y fáciles de encontrar y modificar.

Sin embargo, no todo son ventajas. Lewis (2010) realiza un estudio que compara la programación basada en bloques con lenguajes textuales, utilizando Scratch y Logo con dos grupos homogéneos de estudiantes de último ciclo de primaria. Encuentra que Scratch mejora la comprensión de algunos conceptos de programación con respecto a Logo, pero también que no ocurre en otros, sugiriendo con ello que un lenguaje de bloques no es más efectivo de forma universal que un lenguaje textual.

Garlick y Cancaya (2010) comparan estudiantes de un curso de introducción a la programación universitaria, observando que los estudiantes tradicionales obtienen un mejor rendimiento que los que realizaron el curso en Alice, además de manifestar valores de autopercepción mejores.

También hay abundante literatura comentando que la transición de lenguajes de bloques a lenguajes de texto no es tan suave como podría asumirse. Parsons y Haden (2007) cuestionan el valor pedagógico de Alice para la formación de programación, encontrando que los estudiantes universitarios no realizan conexiones profundas entre su trabajo con Alice y el proceso de codificación

formal. Manifiestan similar preocupación Powers, Ecott y Hirshfield (2007), que encuentran diferencias cognitivas importantes entre las dos modalidades de programación. Es más positivo el estudio de Moskal, Lurie y Cooper (2004), que incluye opcionalmente Alice en una asignatura inicial de programación universitaria, encontrando que los estudiantes que incluyen esa formación mejoran significativamente los resultados finales, siendo este efecto especialmente mayor en estudiantes en "riesgo" (sin experiencia previa de programación o con bajas calificaciones matemáticas). En este caso los estudiantes beneficiados reciben una clase extra que en sí misma puede justificar el efecto.

Entornos posteriores a Alice como Scratch parecen lograr mejores resultados. Estudiantes que han aprendido Scratch en tercer año de secundaria aprenden mejor C# o Java en cuarto, con más confianza y motivación y mejor comprensión de conceptos abstractos (Armoni, Meerbaum-Salant y Ben-Ari, 2015). Wolz y cols. (2009) describen también buenos resultados incorporando Scratch para luego pasar a Java o C en cursos iniciales de programación universitarios, similares a Malan y Leitner (2007).

Una línea de experimentación relevante en este tránsito entre lenguajes visuales y lenguajes de texto es lo que Dann y cols. (2012) proponen en su "transferencia mediada" (*mediated transfer*) entre Alice y Java. Desarrollan un plugin para el IDE NetBeans de Java que permite convertir el proyecto de Alice en código Java, de modo que los conceptos aprendidos con Alice puedan transferirse a un lenguaje de programación a nivel de producción. En su experimentación, los estudiantes que usan ambos lenguajes consiguen una mejora de rendimiento del 10% o superior en todas las secciones del examen final Java (destacable ya que se podría asumir que el tiempo que se invierte en Alice disminuye el tiempo dedicado a Java).

Estos últimos años, muchos cursos iniciales de computación en ingenierías universitarias, en universidades muy representativas como Harvard o Berkeley, utilizan lenguajes de bloques como una aproximación previa o integrada con los lenguajes textuales (Bau y cols., 2017).

Con tantas ventajas como las enumeradas ¿por qué se siguen utilizando lenguajes de texto en lugar de programación visual con bloques en entornos profesionales? Una primera razón es la potencia y la expresividad. El conjunto de simplificaciones que hemos comentado permite un mejor aprendizaje, pero limita a la vez las posibilidades de programar comportamientos específicos. Por ejemplo, un bucle con contador resumido como hemos visto en herramientas de este tipo no permitiría hacer recorridos con dos variables en lugar de una, o decrementando 2 unidades la variable de control. Otra razón es que la manipulación directa tiene muchas desventajas de eficiencia cuando se hacen pequeñas ediciones. Cuando se crea una expresión aritmética como (a/2 + b/2) en un lenguaje de bloques, el programador debe encontrar y arrastrar bloques para los tres operadores

aritméticos, y llenar los huecos de los operandos con variables y números. Si se quiere modificar la expresión a (a+b)/2, debe reorganizarse completamente el bloque de la expresión. Muchos más cambios de los que hacen falta para modificar directamente la expresión textual con teclado. Los investigadores de interfaces de usuario han observado que los lenguajes visuales suelen tener una viscosidad (dificultad para hacer cambios) mayor que los textuales (Green, 1989).

Autores como Meerbaum-Salant, Armoni y Ben-Ari (2011) identifican prácticas habituales de programación poco recomendables, por la manera en la que los aprendices desarrollan sus códigos sin supervisión, o basándose en códigos ya existentes poco adecuados, al menos desde el punto de vista de la buena estructura y legibilidad. Hay otras desventajas de usabilidad de los entornos de bloques con respecto a los lenguajes textuales que reseñan Bau y cols. (2017): baja densidad (los bloques ocupan mucho más espacio de pantalla que sus códigos equivalentes de texto), dificultad de búsqueda y navegación dentro del código, dificultad de control de versiones y sistemas de colaboración.

Estos mismos autores comentan que frente a estas limitaciones, las últimas evoluciones de herramientas de programación basadas en bloques incluyen características orientadas a paliar estos problemas, especialmente en cuanto a usabilidad. Hay dos aproximaciones: entradas de estilo texto y conmutación de modo bidireccional.

1. La entrada de estilo texto, observable en entornos como Greenfoot (Kölling, Brown y Altadmri, 2015) que incorpora un editor basado en marcos gráficos. Ofrece las ventajas de la programación basada en bloques, y a la vez incluye espacios donde puede programarse con texto. Se describen también experimentos de este tipo con GP (Monig, Ohshima y Maloney, 2015). En ambos casos pueden elegirse bloques desde una paleta de selección, pero un programador experimentado puede insertarlos tecleando. En los lugares como las expresiones, donde los bloques necesitan una alta densidad de componentes para poco código, se permite edición de texto tradicional. Se mantiene la posibilidad de interacción directa de ratón para gestionar la estructura general y las declaraciones de alto nivel.

2. Conmutación de modo bidireccional, como en Pencil Code (Bau y cols., 2015) con Cofeescript o JavaScript, App Lab[30] de code.org con JavaScript, BlockEditor con Java (Matsuzawa y cols., 2015) o Tiled Grace (Homer y Noble, 2014) con Grace. La hipótesis en estos casos es que los usuarios pueden beneficiarse de la eficiencia del modo textual, a la vez que utilizar las ventajas comentadas del modo visual. En esta serie de entornos puede conmutarse en cualquier momento entre una vista de bloques y otra de texto del mismo programa (figura 2.7), manteniéndose una correspondencia continua que informa en el caso del texto de cualquier error léxico o de sintaxis que pueda producirse para evitar que el código pierda su corrección.

[30] https://code.org/educate/applab

Si se cometen errores en el código de texto, se evita la conversión hasta que se solucionen o se marcan de una forma especial en el código de bloques.

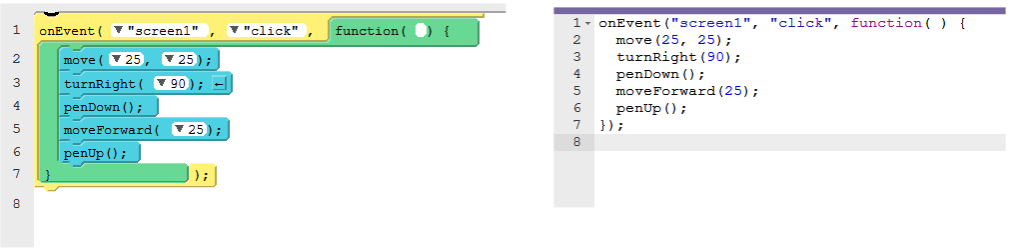


Figura 2.7. Código bidireccional en App Lab: bloques a la izquierda, JavaScript a la derecha.

Aunque no es el ámbito fundamental de nuestra investigación, destacamos que las posibilidades de los lenguajes visuales basados en bloques están posibilitando su uso creciente en muchos otros contextos tecnológicos no ligados con el aprendizaje. Es el caso de modelado 3D para su uso en impresoras 3D y otros dispositivos (Johnson y Bui, 2015; Koschitz y Rosenbaum, 2012), el manejo de drones (Tilley y Gray, 2017) o las consultas en web semántica (Bottoni y Ceriani, 2015). También se encuentran disponibles *frameworks* para crear soluciones de dominios específicos en base a lenguajes basados en bloques configurables. Es el caso de Blockly[31], Droplet (Bau, 2015) o el primer lenguaje de definición de lenguajes de bloques, OpenBlock (Roque, 2007). OpenBlock está basado en Java, es la base de StarLogo TNG y ha sido usado para otros entornos como App Inventor Classic y BlockEditor.

2.4. Aprendizaje basado en juegos

Observamos que en nuestro repaso por herramientas y estudios que facilitan el aprendizaje del PC, hay una gran abundancia de juegos o entornos que permiten programarlos. Con objeto de contextualizar mejor las herramientas empleadas en nuestra investigación, realizamos una revisión de las posibilidades educativas del juego, de cara a enlazarla con su importancia con el aprendizaje del PC.

El aprendizaje basado en juegos (GBL, *Game Based Learning*) es un área significativa dentro del aprendizaje, que ha cobrado más relevancia en las últimas décadas. Wu y cols. (2012) realizan un metaanálisis en el que observan que el GBL se relaciona con numerosos fundamentos de aprendizaje, y aunque los estudios no son homogéneos en cuanto a la relación con esos fundamentos, usar simulaciones interactivas y juegos reportan ganancias cognitivas superiores y mejores actitudes hacia el aprendizaje que sus equivalentes tradicionales (Vogel y cols., 2006). Una de sus variantes más significativas es el aprendizaje basado en juegos digitales (DGBL,

[31] https://developers.google.com/blockly

Digital Game Based Learning) (Prensky, 2003), que es la que tiene relación más directa con la mayoría de las herramientas que nos ocupan.

No hay ninguna razón de base para que el aprendizaje del PC tenga una relación directa con el juego, pero el público objetivo lo justifica especialmente. Papastergiou (2009) compara una actividad digital con la misma actividad presentada como un juego en un colectivo de adolescentes y encuentra una mayor motivación y un mejor rendimiento educativo. Es un hecho que los juegos están muy presentes en la mayoría de las herramientas relacionadas y dos enfoques son los más populares para facilitar el aprendizaje del PC: 1) aprender a través del ejercicio de diseñar juegos y 2) aprender a través del juego.

2.4.1. Aprender diseñando juegos

Aprender diseñando juegos ha sido explorado en multitud de formas. Utilizando herramientas de autoría de videojuegos como GameMaker (Kuruvada y cols., 2010; Eow y Baki, 2010), con motores de juego específicos como Neverwinter Nights, que permite generar juegos RPG en tres dimensiones (Robertson y Howells, 2008), con diferentes motores en cada edición (Seif El-Nasr y cols., 2007) o con herramientas específicas de PC: Alice (Lee y cols., 2011), AgentSheets (Basawapatna, Koh y Repenning, 2010), Stagecast Creator (Denner, Werner y Ortiz, 2012), y por supuesto Scratch (Wilson, Hainey y Connolly, 2012).

También, especialmente en niveles universitarios, se puede realizar la creación planteando fases progresivas de evolución del juego, de manera muy similar a como se suelen plantear en aprendizaje basado en proyectos (PBL, *Project Based Learning*) como describen Sung y cols. (2010).

Begel y Klopfer (2007) defienden la importancia del diseño de juegos como elemento clave del aprendizaje del PC, su capacidad inclusiva y las posibilidades que abren los entornos 3D para no limitarse a juegos solo en dos dimensiones. Según ellos, el diseño y la construcción de juegos requiere y facilita el aprendizaje de habilidades de programación, al incluir los juegos muchas conexiones naturales con los programas, como caracteres animados y sus comportamientos, interacción humana, estado de juego (y su relación con estructuras de datos), personajes controlados por la computadora (y su relación con la IA). El diseño de un juego además exige conocer y aplicar habilidades como secuenciación, diferenciación y repetición de operaciones. La escenografía de un juego engloba facetas artísticas y funcionales. En las necesidades de diseño de los juegos se encuentran multitud de relaciones con las matemáticas, estadística, física y otras áreas STEM.

Uno de los planes más ambiciosos es el diseño de juegos escalable (SGD, *Scalable Game Design*) que describen Repenning y Ioannidou (2008). El objetivo es incluir en el currículo educativo de secundaria en EEUU un programa de educación en PC, a través del diseño de juegos y una serie de simulaciones STEM. Tras probarlo con

miles de estudiantes, los autores describen que la aproximación es positiva y que aumenta la participación incluso en los contextos sociales más desfavorecidos.

Ampliando esta propuesta se plantean las zonas de flujo próximo (Basawapatna y cols., 2013). Este modelo pedagógico combina las bases psicológicas del concepto de flujo de Czikszentmihalyi (1990) con la zona de desarrollo próximo de Vygotsky (1996). Primero se presenta el proyecto a los estudiantes, lo que los lleva a la zona de desarrollo próximo que es la zona ideal para el aprendizaje. En ella el aprendiz se encuentra en la frontera de su conocimiento. Con el soporte adecuado de los formadores, los estudiantes pueden traspasar esta frontera y aprender nuevos conceptos como los que se trabajan en el PC. Los autores diseñan un camino de aprendizaje dentro del proyecto de diseño de juegos escalable, que discurre en una serie de juegos de complejidad creciente, desde *arcades* de los 80 como Frogger hasta juegos recientes como los Sims. Según los autores esta metodología resulta altamente efectiva.

Denner, Werner y Ortiz (2012) identifican la correspondencia entre los juegos que desarrollan chicas de secundaria y una serie de conceptos clave de programación informática. Encuentran que la programación de juegos puede ser efectiva en motivar a aprendices adolescentes en niveles moderados de actividad de programación compleja, niveles moderados de usabilidad, pero solo niveles bajos de organización y documentación de código. Wilson, Hainey y Connolly (2012) adaptan el esquema de código y analizan el nivel de desempeño de 29 programas de Scratch en primaria. La interacción de usuario, los bucles y las condicionales aparecen como los conceptos más comunes, y los números aleatorios, entre los menos comunes.

2.4.2. Aprender a través del juego

El segundo enfoque, aprender PC a través del juego, tiene una creciente aplicación en los últimos años con el crecimiento de las herramientas disponibles. El primer entorno que aborda este enfoque es Karel el Robot (Pattis, 1981) donde el estudiante debe programar un robot que puede detectar paredes y realizar acciones de movimiento. Aunque se ha ido mejorando la usabilidad e incorporando programación de bloques, muchas de las actividades posteriores han sido influenciadas por Karel.

Se encuentran muchos casos de juegos orientados al PC en el marco de la enseñanza universitaria de programación. Así encontramos Robocode (Hartness, 2004), en ese caso utilizando la competición como motivador asociado al juego. En Robocode los jugadores diseñan y programan la IA de su robot, que se enfrenta con ella a otros jugadores. Long (2007) investiga los factores que mantienen a los estudiantes jugando a Robocode, desarrolla los motivadores implícitos que tiene el componente de juego y concluye que estaban más motivados con descubrir

algoritmos que con programarlos o depurarlos. Incluso conceptos más complejos como la recursividad se pueden introducir con juegos de PC, como describen Chaffin y cols. (2009) en un juego que consta de tres puzles de programación basados en recorridos de estructuras de datos. Muratet, Torguet y Jessel (2009) utilizan un motor de juegos de estrategia multijugador, para diseñar código que controle el comportamiento de las unidades que maneja, de nuevo en un enfrentamiento competitivo contra otros jugadores. Estos autores destacan la importancia de que esta aproximación cuente con tutores que asistan a los estudiantes mientras diseñan sus códigos, especialmente con aquellos con menor ritmo de aprendizaje.

A pesar de las ventajas de usar el proceso de juego para el aprendizaje del PC, hay inconvenientes y peligros que es importante considerar. Cuando se utiliza la competición, hay que valorar en qué medida puede causar el efecto contrario al deseado. Aunque hay estudios que aconsejan la competición por su capacidad de diversión y motivación (Ladd y Harcourt, 2005), las características psicológicas, sociales y culturales pueden provocar que algunos estudiantes abandonen o se sientan desmotivados o con exceso de presión. También es importante cómo se diseñen los juegos competitivos, ya que la diferencia de ritmo de aprendizaje o de velocidad de programación puede influir en los resultados de la competición.

Jesse Schell (2008) explica que la percepción de lo que es un juego puede cambiar mucho de una persona a otra. Sin embargo, una característica que parece coincidir en todas las percepciones de los juegos es la resolución de problemas. De hecho, define los juegos como "una actividad de resolución de problemas, observada con una actitud de juego". Schell también sugiere que un juego debe generar nuevos problemas según avanza. Este es un tema clave en los juegos diseñados para aprender PC, cómo entrelazan los contenidos de aprendizaje con los retos que se plantean.

Varios estudios proponen una serie de contenidos en el juego solo ofreciendo conocimiento abstracto, sin resolución de problemas (Papastergiou, 2009; Yeh, 2009). En estos casos, los estudiantes progresan en el juego cuando contestan correctamente a una serie de cuestiones de comprensión, de manera muy similar a muchos de los juegos de preguntas y respuestas que en los últimos años se usan para ludificar las clases. Si bien esta posibilidad tiene efectos positivos, corre el riesgo de carecer de integración entre la experiencia de aprendizaje y la de juego, con lo que el estudiante puede concentrarse en jugar en vez de en aprender del juego, convirtiendo el aprendizaje constructivista en uno instructivista y, en ocasiones, no afectando directamente a la mejora de los resultados del aprendizaje (Kazimoglu y cols., 2011). Graven y MacKinnon (2008) proponen un juego muy elaborado de tipo MMORPG (juego de rol masivo multijugador en línea, *Massively Multiplayer Online Role-Playing Game*), en el que los estudiantes navegan en un mundo virtual conversando con jugadores reales y computacionales, enfrentándose

a preguntas sobre conceptos del PC (sintaxis y particularidades de métodos de Java). Con todos los esfuerzos de desarrollo de juego, los propios autores destacan las limitaciones y las mejoras necesarias para que se establezca mejor relación entre los materiales a aprender y el desarrollo del juego.

Es importante establecer una clara distinción entre estos juegos que soportan y refuerzan el conocimiento conceptual, que debe aprenderse fuera del juego, de aquellos sistemas que, dentro del mismo juego, incluyen el conocimiento procedimental y aplicado al objetivo del aprendizaje. Estos segundos son los que más se han desarrollado en esta última década de forma asociada al PC.

Otro de los problemas de diseño de los juegos de aprendizaje del PC es la carencia de *feedback*. Es importante tener mecanismos claros de *feedback* para ayudar a los aprendices a aprender de sus errores mientras juegan, para dirigirles hacia buenas prácticas mientras programan. Incluso en los entornos de programación visual de bloques, se encuentra poco *feedback* de las estrategias adecuadas, o de las implicaciones de resolver un problema en un nivel abstracto para la reusabilidad. No solo hay que retroalimentar lo que funciona o no, sino cuándo la solución es mejorable, como una solución lineal que podría ser estructurada, o una solución ad hoc que podría ser generalizable. Aunque esto es un problema que puede ocurrir también en la enseñanza convencional, es especialmente reseñable cuando el aprendizaje ocurre con un juego, que precisamente es conveniente porque puede dar *feedback* instantáneo y efectivo (Kazimoglu y cols., 2011). La investigación muestra que después de la diversión el segundo factor de motivación en los juegos educativos para los aprendices es el *feedback* significativo (Eagle y Barnes, 2008). Similarmente, Gee (2003) destaca los retos, un conjunto claro de objetivos, *feedback* efectivo y la emoción de superar los retos como los factores principales que motivan a los jugadores en juegos educativos.

Un último problema de diseño se relaciona con las experiencias y herramientas educativas que se enfocan en enseñar cómo programar código en lugar de cómo desarrollar el PC. Es importante considerar que el reto de la educación del PC no es si los estudiantes pueden resolver un problema en un lenguaje de programación, sino si los estudiantes pueden encontrar soluciones a retos computacionales utilizando conceptos del PC. Repenning, Webb y Ioannidou (2010) afirman que las soluciones presentadas en entornos GBL no siempre pueden unirse conceptualmente al origen del problema, ya que las soluciones normalmente se encuentran en más bajo nivel que el concepto asociado del PC. Mencionan que es esencial no atrapar a los estudiantes solo en actividades con lenguajes de juguete, que pueden conseguir un breve impacto de entusiasmo, si fallan en el objetivo principal de ayudarles a progresar en la motivación hacia aplicaciones STEM.

Además de los estudios ya citados, hay multitud de aproximaciones al PC utilizando el juego como metodología de base. Algunos son estudios que evalúan los

comportamientos de los aprendices además de su motivación en el aprendizaje de programación. Liu, Cheng y Huang (2011) crean un juego de simulación (basado en un tren y su circulación por una serie de vías con dispositivos de control) y analizan la respuesta y comportamiento de resolución de problemas de 110 estudiantes durante el juego. Encuentran que los estudiantes motivados por el juego usan con frecuencia estrategias analíticas para descubrir las soluciones posibles y que también exploran maneras de aplicarlas. No obstante, los estudiantes aburridos por el juego solo resuelven problemas en un nivel superficial.

En esta línea de intentar incorporar en la mecánica de un juego aspectos del PC, encontramos también la experiencia de Holbert y Wilensky (2011) que diseñan un juego de carreras, FormulaT, que permite interactuar a escolares de primaria y secundaria con colores que manipulan los comportamientos dinámicos de su coche en una carrera. Los autores identifican esta actividad con lo que llaman "estrategias computacionales sistemáticas", y comprueban como dentro de la actividad de juego los aprendices utilizan diferentes habilidades incluidas en el PC como ordenación, repetición o depuración.

Kazimoglu y cols. (2012) proponen un modelo de juego para el aprendizaje del PC, "Program your Robot", un robot moviéndose por un espacio en base a órdenes de código, y estudian los conceptos del PC relacionados con ese modelo y cómo el juego se relaciona con ellos: resolución de problemas, pensamiento algorítmico, depuración, simulación y socialización. Los estudiantes encuentran positiva la aproximación para el aprendizaje del PC.

2.5. Conclusiones

A lo largo de este capítulo, hemos realizado una revisión de las áreas clave en las que se desarrolla nuestra investigación. Queremos experimentar con una herramienta que permita a una serie de aprendices llevar a cabo sus primeras experiencias con el PC. La actividad de programación con esa herramienta permitirá elaborar su pensamiento algorítmico y trabajar los otros aspectos esenciales del PC, utilizando un entorno visual basado en bloques. Nos planteamos también si la falta de explicitación de la habilidad de generalización en las herramientas encontradas permite que ese sea un aspecto relevante a tener en cuenta.

Necesitamos que esa herramienta permita el registro de grano fino de interacción de los aprendices para poder realizar analítica del aprendizaje. Muchos estudios mencionados se basan en muestras pequeñas o demasiado homogéneas (a menudo de un solo centro educativo). Nos planteamos la importancia de afrontar esas limitaciones con la utilización de una muestra grande y diversa.

Para que la investigación sea abordable y la prueba sea autónoma, planteamos un enfoque de aprendizaje basado en juegos, y en particular aprendiendo a través del juego.

Hacemos algunas reflexiones sobre las herramientas existentes, sobre todo para decidir qué aspectos tener en cuenta en nuestro estudio. Hemos visto un número sustancial y creciente de opciones accesibles para aprendices digitales del PC. Un camino de aprendizaje interesante parece ser empezar con algunos de los juegos de retos durante unos días/semanas, y pasar desde ahí a herramientas basadas en lenguajes de programación visual, que pueden llevar meses o años de actividad. La diversión y motivación es muy asequible y hay diversidad de opciones, además de temáticas que hacen buen uso de franquicias y personajes conocidos del mundo del cine o los videojuegos para reforzar la experiencia.

No obstante, hay algunas limitaciones que considerar para la siguiente generación de juegos relacionados con el PC. En primer lugar, hay un uso excesivo de primitivas de acción que tienen que ver con el movimiento y la rotación ortogonal (influidos como estamos, quizás, por Logo y su importancia histórica en este campo), y se aprovechan poco otras opciones visuales, auditivas, o rítmicas, que permiten también el desarrollo de pensamiento algorítmico a la vez que exploran otras habilidades diferentes a la lateralidad y a la visión espacial. En este sentido, anotamos la relevancia de algunas de las más recientes actividades incorporadas en Code.org y Made with Code, con elementos multimedia.

Encontramos una amplia predominancia de bloques. Herramientas como Kodu que usan diagramas de flujo abren posibilidades para diseñar nuevas herramientas de PC que combinen otras expresiones visuales complementarias a la mayoritaria abstracción visual usada: bloques anidados (en algunos casos, sobre todo para las edades inferiores, iconos más que bloques).

Otra carencia generalizada en este campo es el acceso a la información de uso. Es especialmente importante en este campo naciente poder investigar cómo se comportan los aprendices y cómo los sistemas facilitan su aprendizaje, con lo que se echa en falta la distribución de información de acceso abierto para su análisis.

Una ausencia final que notamos es el aprendizaje adaptativo. Prácticamente todas las herramientas se comportan siempre de la misma forma, independientemente de la edad, conocimiento previo, o habilidades mostradas por el aprendiz. En un contexto en que la actividad del usuario puede ser registrada, es especialmente interesante y viable que se utilice para adaptar y mejorar la experiencia educativa.

Analizando solo la categoría de juegos basados en retos de programación, observamos la importancia de generar más implicación después de que los retos se han acabado, incorporando técnicas de ludificación ya conocidas como puntuación, clasificaciones, objetivos mejorados, y la incorporación de niveles creativos en los que el reto no está limitado por retos cuantificables simples. Esta es una línea en la que las entidades con más recursos, como Code.org o Tynker, ya están trabajando. Otro aspecto clave en estos sistemas es la estructuración secuencial de los contenidos (andamiaje, *scaffolding*). Además del esfuerzo obvio del diseño de

niveles (una necesidad compartida por educación y juegos), basado sobre todo en la experiencia de los diseñadores, es un reto sistematizar el proceso que asegure la progresión adecuada del aprendizaje y la motivación, en búsqueda del flujo que los juegos persiguen (y en muchos casos consiguen). También es necesario investigar las pautas a seguir en la generación automática de niveles/retos, en la línea conocida en muchos videojuegos de generación procedimental.

Otro aspecto relevante es la construcción de las introducciones y tutoriales, para maximizar el objetivo de aprendizaje que deberían perseguir estas herramientas. Identificar el tipo de pensamiento que el aprendiz está utilizando para resolver cada problema progresivo es un elemento clave, como mencionan Touretzky, Gardner-McCune y Aggarwal (2017) utilizando Kodu.

Una carencia final en juegos basados en retos es que pocos sistemas permiten a los participantes o supervisores la creación de nuevos retos. Esto limita la experiencia y cierra prematuramente el ciclo de aprendizaje, dejando al aprendiz solo en el lugar de usuario sin poder ponerse en el papel de creador. Herramientas como MakeWorld o Robozzle, que no permiten solo jugar sino también editar nuevos mundos, marcan este camino hacia la siguiente generación de juegos que incrementen la personalización de retos a través de la modificación o la contribución creativa, tan característica en el mundo de la remezcla, ya habitual en las actividades basadas en lenguajes de programación.

Al respecto de esa categoría de lenguajes de programación, hay limitaciones intrínsecas a los entornos visuales basados en bloques comparados con los lenguajes basados en texto. En proyectos de programación grandes esto se percibe rápidamente, debido a las limitaciones de visibilidad de código, dificultad de navegación por el código, o falta de control sobre las modificaciones realizadas (Bau y cols., 2017).

La conversión bidireccional entre bloques visuales y lenguaje de programación textual, como hace App Lab de Code.org, es una opción relevante para el aprendizaje. Esta característica permite a los aprendices elegir la vista más útil, dependiendo de la complejidad y alcance del proyecto. Hacemos notar que las plataformas orientadas a juego también están incluyendo cuadros de mando (*dashboards*) para profesorado (Code.org, Kodable, Tynker), que todavía son muy limitados o no existen en las plataformas orientadas a lenguaje de programación, reflejando la carencia de herramientas de evaluación ya mencionada.

También hemos visto en nuestro análisis una pequeña muestra de una tercera categoría, los videojuegos que incorporan en sus mecánicas programación visual y en ese sentido permiten desarrollar PC de un modo lúdico. En este campo Minecraft es un ejemplo muy interesante para diseñadores de juegos: buenos videojuegos pueden ser diseñados considerando mecánicas específicas de PC, lo que puede ser un campo de innovación para el sector del diseño de juegos. Más

allá de fabricantes particulares, empieza también a haber iniciativas de herramientas de autoría que permiten desarrollar juegos de contextos diversos que puedan ser utilizados para educación de PC con programación basada en bloques, como presentan Su y cols. (2019) con tecnología Blockly de base.

Tras todo este análisis, determinando que no podemos abordar todos los aspectos que nos gustaría en una sola investigación, hemos seleccionado un conjunto de características, las que más nos interesan de cara a nuestra experimentación. No encontrar ninguna herramienta que las satisfaga todas es la razón fundamental por la que desarrollamos nuestras propias herramientas, que presentamos en los siguientes capítulos. Presentamos la tabla 2.2 que muestra estas características y a título informativo incluye nuestras dos herramientas Kodetu y KodetuGen2.

Herramienta	Rep	Alt	Dep	Disp	Blq/Ico	Gen	Txt	Log	Reg	Grp	Fac	Etap	Web	Plugin
Bee-Bot				no	iconos						X	Pre		
Blockly Games	X	X	X	Ver	bloques		X				X	Prim/Sec	X	
Cargo-Bot		X	X	Hor	iconos						X	Prim		
Code.org	X	X	X	Ver	bloques		X		X	X	X	Prim/Sec	X	
Daisy	X		X	Ver	bloques						X	Pre/Prim		
Kodable	X	X	X	Hor	iconos				X	X	X	Pre/Prim	X	
Kodetu	X	X	X	Ver	bloques		X	X	X		X	Prim/Sec	X	
KodetuGen2	X	X	X	Ver	bloques	X	X	X	X	X	X	Prim/Sec	X	
Lightbot			X	Hor	iconos							Pre/Prim	X	X
Made w/Code	X	X	X	Ver	bloques							Prim/Sec	X	
SpriteBox	X		X	Ver	iconos		X					Pre/Prim	X	X
The Foos	X	X	X	Hor	bloques						X	Pre/Prim	X	
Tynker - Act	X	X	X	Ver	bloques				X	X		Prim/Sec	X	

Tabla 2.2. Características relevantes para nuestro estudio de las herramientas revisadas: Rep=Estructuras repetitivas, Alt=Alternativas, Dep=Depuración visual en ejecución, Disp=Disposición (vertical/horizontal), Blq/Ico=Bloques o iconos, Gen=Generalización, Txt=Lenguaje texto equivalente, Log=Registro de grano fino, Reg=Registro de usuario, Grp=Definición de grupos, Fac=Facilidad de instalación/uso, Etap=Etapas formativas recomendadas (Pre, Prim, Sec), Web=Tecnología Web, Plugin=Necesita plugin.

There were rags of many colors
Every piece was small
And I didn't have a coat
And it was way down in the fall
Mama sewed the rags together
Sewing every piece with love
She made my coat of many colors
That I was so proud of

Dolly Parton. "Coat of Many Colors"

Capítulo

3

Herramienta Kodetu

Hemos reseñado el creciente número y variedad de herramientas existentes para trabajar el PC en aprendices y cómo se ha realizado la evaluación del aprendizaje del PC de distintas formas. En base a ello, consideramos importante realizar un estudio pormenorizado de la respuesta de los aprendices en este tipo de iniciativas para poder analizar sus procesos de aprendizaje y obtener conclusiones científicas.

Para ello, desarrollamos en 2014 un sistema nuevo para el aprendizaje de principios muy básicos del PC en el área de algoritmia básica, utilizable en experiencias cortas (20-90 minutos) por aprendices de cualquier edad, que posibilitara almacenamiento detallado de todas las interacciones realizadas para poder analizarlas con posterioridad de forma sistemática. Planteamos además recoger un conjunto de participantes suficientemente grande (más de 3.000 aprendices) y diverso (más de 100 grupos de más de 15 centros escolares).

3.1. Objetivos de investigación

En este punto de nuestro estudio, queríamos entender los procesos de aprendizaje del PC en aprendices de programación, en particular estudiantes de educación primaria y secundaria (especialmente en el rango de edad de 8 a 14), usando un entorno *online* basado en retos, presentado a modo de juego de superación de niveles. Para ello, diseñamos y desarrollamos un sistema al que llamamos Kodetu, lo propusimos tanto a escolares como a público general y analizamos sus registros, con las siguientes preguntas de investigación en perspectiva:

PI_K1 ¿Cuál es la influencia de las características personales (sexo, edad, afinidad tecnológica y conocimiento de programación previo) en las primeras experiencias del PC?

PI_K2 ¿Cuántas soluciones diferentes encuentran los aprendices en sus primeros retos de programación y qué se puede concluir de estas diferencias?

PI_K3 ¿Qué variables podrían ayudar a plantear estrategias de personalización y adaptación de este tipo de herramientas de aprendizaje del PC?

3.2. Metodología

3.2.1. Diseño de Kodetu

Para responder a nuestras preguntas de investigación, desarrollamos Kodetu[32], una plataforma *online* basada en retos para el aprendizaje de algunos de los conceptos más básicos de programación relacionados con algoritmia del PC: sentencias básicas, estructura de secuencia, estructura repetitiva y estructura alternativa.

Kodetu está basado en Blockly, una librería software desarrollada por Google para crear entornos de programación visual basada en bloques. Particularmente en *Maze*, una de sus variantes, que implementa una serie de niveles de laberinto del que hay que encontrar la salida utilizando bloques visuales de código. Modificamos esta actividad de tres maneras diferentes:

1) Aumentamos el número de niveles hasta 15.
2) Incorporamos un sistema de registro detallado que recoge las acciones de los aprendices mientras usan la plataforma. Basamos el proceso en registros de grano fino de la literatura como el indicado por Vihavainen, Luukkainen y Ihantola (2014).
3) Estructuramos un sistema de variación de algunas características generales, para posibilitar variantes del juego con grupos diferentes de aprendices.

Al inicio de la sesión, todos los aprendices son informados de las características de la investigación y de que su participación en el estudio es voluntaria. Para evitar un sistema de registro que es especialmente delicado en menores de edad que en muchos casos no pueden tener aún correo electrónico personal, definimos un sistema de identificación oculto que asigna un código alfanumérico aleatorio único a cada aprendiz. Además, no se piden datos de identificación personal ni se guardan direcciones IP, con lo que la privacidad y el anonimato están garantizados.

El sistema sí solicita en la entrada datos demográficos (edad, sexo y ciclo educativo), nombre del centro educativo, información de si se tienen conocimientos previos de programación (sí/no) y gusto por la tecnología (valor numérico de 1 -bajo- a 10 -alto-). Seguidamente, se muestra una breve encuesta que recoge en una escala Likert de siete opciones la percepción del aprendiz sobre 1) la tecnología: fascinante/corriente, atractiva/poco atractiva, emocionante/poco emocionante, poco importante/importante, aburrida/interesante; y 2) una carrera en STEM, con la misma escala en las mismas cinco dimensiones.

[32] http://kodetu.org

Tras el consentimiento informado y la introducción de dichos datos, Kodetu muestra un laberinto en forma de camino ortogonal de una esquemática estación espacial (en un tablero virtual de 8x8, figura 3.1). Un astronauta se encuentra localizado en un punto de ese laberinto y una marca de meta única se sitúa en otro lugar del mismo. El astronauta debe caminar sobre la superficie de esa estación espacial para encontrar la meta, evitando caerse al espacio (salirse del camino).

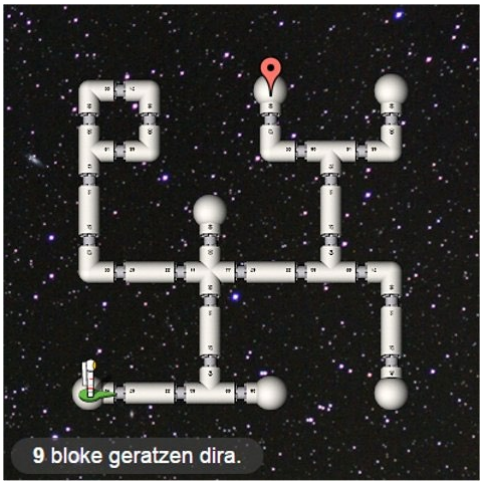


Figura 3.1. Laberinto ejemplo de Kodetu.

Los aprendices no pueden mover al astronauta de forma directa (con teclado o interacciones de ratón). En su lugar, deben definir los movimientos del astronauta usando unos pocos bloques de código (figura 3.2, bloques violetas) que representan movimientos básicos:

- Movimiento hacia adelante
- Girar a la izquierda
- Girar a la derecha

Figura 3.2. Ejemplo de código formado por bloques en Kodetu. Los bloques violetas representan instrucciones simples de movimiento del astronauta. Los verdes, estructuras repetitivas. Los azules, estructuras alternativas. Nótese que las estructuras complejas como la condicional y la repetitiva utilizan un solo bloque de Kodetu, aunque se representan visualmente en varias líneas. Obsérvese también el borde naranja que se utiliza en la vista de ejecución de cada bloque.

Posteriormente en el juego aparecerán otros bloques adicionales:

- Comprobar si hay camino adelante, a izquierda o derecha
- Realizar unos bloques (acciones) u otros dependiendo de esas comprobaciones (estructura condicional), con dos formatos: condicional de una rama y de dos ramas.
- Repetir bloques hasta que se encuentra la meta.

La interfaz completa que usará el aprendiz presenta una zona principal de la aplicación dividida en tres espacios: en la parte izquierda se muestra el laberinto a resolver, en la columna central los bloques disponibles para programar y en la zona derecha se representa el espacio de código o espacio de trabajo (figura 3.3).

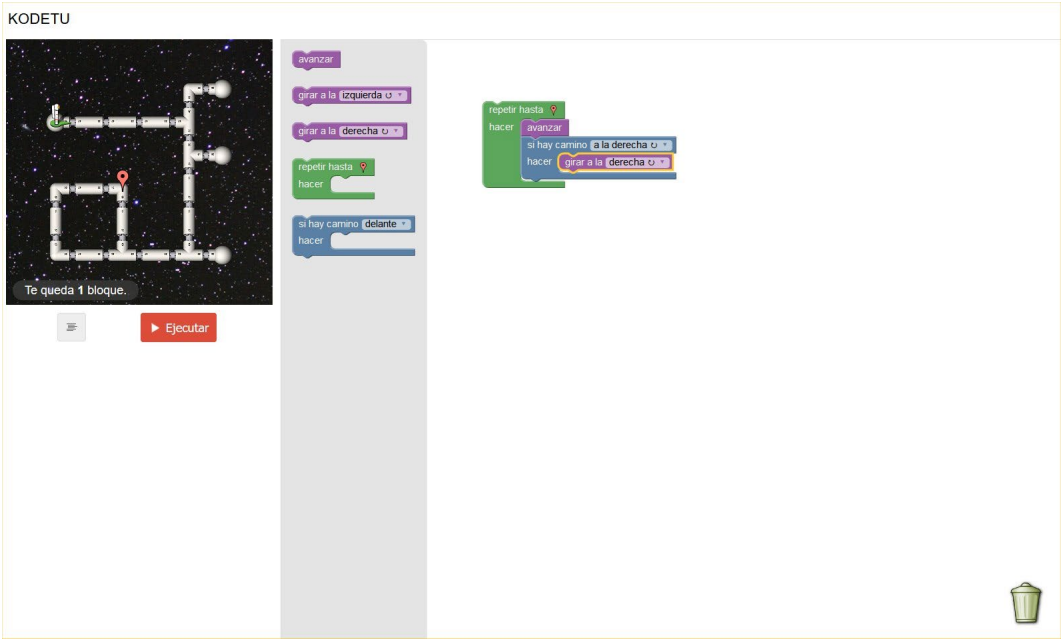


Figura 3.3. Aspecto general de la interfaz de uno de los niveles de Kodetu.

El aprendiz puede mover con el ratón, pulsando y arrastrando, bloques libremente del espacio central al de trabajo (se duplican cada vez que se sacan del espacio central, con lo que pueden utilizarse repetidas veces), o retirarlos del espacio de trabajo a la papelera (o arrastrándolos a la izquierda). Dentro de este espacio de código, el aprendiz puede además unir unas piezas a otras utilizando su forma. De manera similar a otras herramientas de programación visual, los bloques de programación tienen una forma de pieza de puzle con pequeños salientes/entrantes que determinan las posibilidades (limitadas) en las que unas piezas se pueden conectar con otras. En Kodetu la estructura repetitiva no tiene enganches posteriores (es decir, es la última que se puede poner en una construcción de código) y el resto de bloques pueden unirse libremente en secuencia vertical.

Tras colocar las piezas, cuando el aprendiz estima que tiene la solución o en cualquier momento en el que quiera probarla, puede pulsar el botón rojo "Ejecutar", que hace que el astronauta realice los movimientos que están representados en los bloques del espacio de trabajo, de una forma animada en pantalla, con un sencillo efecto de iluminación de borde naranja que muestra el bloque que se está ejecutando en cada movimiento del astronauta.

Si el resultado de la ejecución es incorrecto (el astronauta no llega a la meta, se "cae" al espacio, o entra en un bucle infinito), se muestra un mensaje de error y el aprendiz puede cambiar su código y reintentar la ejecución.

Si el astronauta llega a la meta, se informa del éxito en el nivel y aparece un nuevo reto (nivel superior) donde la complejidad del juego se incrementa; en algunas ocasiones, porque el laberinto es mayor o más difícil; en otras, porque se introducen nuevos conceptos de programación. Diseñamos el juego Kodetu con 15 niveles de complejidad incremental que describiremos en breve.

Según los aprendices juegan a Kodetu, se guarda un registro de todos los cambios en el espacio de trabajo con lo que obtenemos en nuestra plataforma una detallada base de datos compuesta por todos los registros que muestran la progresión de cada aprendiz en los distintos retos propuestos. La información incluida en cada entrada de registro contiene la marca temporal, el nivel jugado, el código contenido en el espacio de trabajo en ese momento y la información de resultado del intento de resolver el reto (actualizada en la pulsación del botón de ejecución), con 5 posibilidades: 1) sin intento (aún no se ha pulsado el botón de ejecución), 2) éxito (el astronauta llega a la meta), 3) fallo (el código acaba sin que llegue a la meta), 4) caída (cae al espacio) o 5) bucle (se repite indefinidamente el camino sin resolverse el laberinto).

Kodetu propone un flujo de aprendizaje progresivo a través de los 15 niveles de reto que los aprendices deben resolver. El nivel de complejidad de estos niveles progresa del siguiente modo.

El nivel 1 (figura 3.4) presenta un reto trivial a modo de tutorial de funcionamiento del sistema. Podemos observar cómo se representa el astronauta con una flecha verde que indica la dirección en la que está mirando, muy importante porque marcará su movimiento. El espacio de trabajo se inicia con una pieza "adelante" (a partir de ahora, **ad**) con la que el aprendiz tiene solo que juntar otra pieza equivalente de avance para que el astronauta llegue en dos pasos al objetivo (indicado con un marcador rojo, en base al convencionalismo gráfico de marcado geográfico popularizado por Google Maps). Se muestra una pista inicial con el texto *"Une un par de bloques de 'avanzar' para ayudarme a llegar a la meta"* y otra *"Ejecuta tu programa para ver qué pasa"* después de mover la pieza. Este

nivel y los siguientes tienen solo disponibles las tres piezas de movimiento directo: la mencionada **ad**, la de giro a la izquierda (en adelante, **gi**) y la de giro a la derecha (en adelante, **gd**). Hasta el nivel 7 solo podrán formarse secuencias que combinen esos tres bloques.

Figura 3.4. Nivel 1 de Kodetu.

Los niveles 2 y 3 presentan los giros a izquierda y derecha, para que el aprendiz aprenda cómo funcionan los bloques de rotación y cómo esta rotación es relativa a la dirección en la que mira el astronauta (figura 3.5). Las soluciones esperables son, respectivamente, `[ad ad gd ad]` y `[gd ad ad]`.

Figura 3.5. Laberintos de niveles 2 (izda.) y 3 (dcha.) de Kodetu.

Los niveles 4 y 5 (figura 3.6) requieren combinar al menos dos rotaciones, con distintas orientaciones para que el aprendiz reconozca la lateralidad de giro desde la perspectiva del astronauta, diferenciándola del punto de vista convencional. Soluciones posibles: `[gd gd ad ad ad]` y `[ad gi ad gd ad]`.

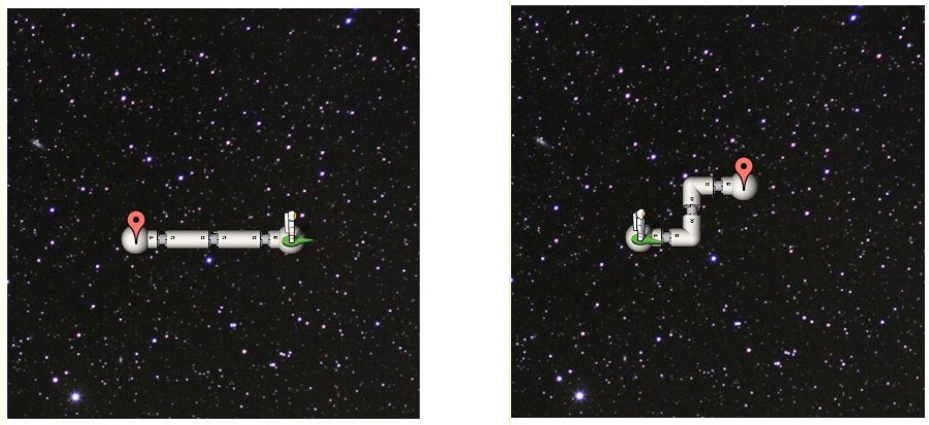


Figura 3.6. Laberintos de niveles 4 (izda.) y 5 (dcha.) de Kodetu.

El nivel 6 (figura 3.7, izquierda) necesita combinar tres giros y tres fases de avance de distintas longitudes, con el código esperable `[gd ad gi ad ad ad gi ad]`. El nivel 7 (figura 3.7, derecha) es un laberinto más largo, con la intención de mostrar la cantidad de trabajo manual que requiere realizar una ruta larga paso a paso. Hacen falta al menos 15 bloques para resolverlo: `[ad ad ad ad gd ad ad ad ad gd ad ad gd ad ad]`.

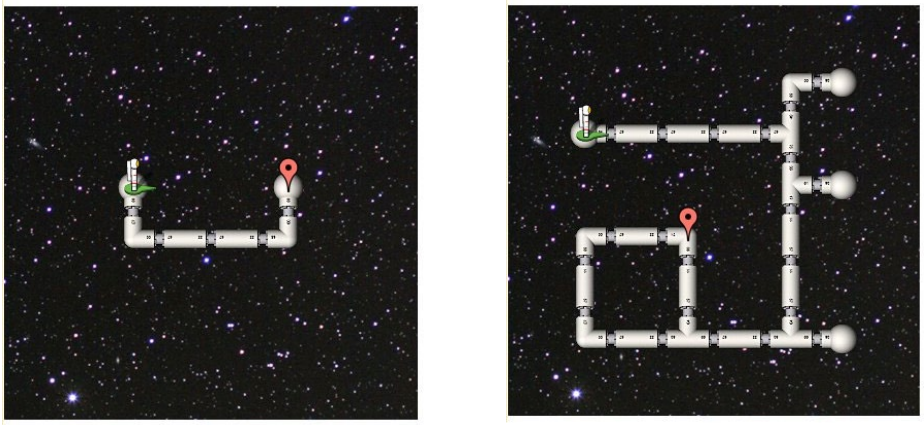


Figura 3.7. Laberintos de niveles 6 (izda.) y 7 (dcha.) de Kodetu.

El nivel 8 (figura 3.8, izquierda) introduce los bucles añadiendo el bloque *repetir* (en adelante `rep`), que permite ejecutar una serie cualquiera de otros bloques de forma repetida hasta que se llega a la meta. Además, este nivel limita por primera vez el tamaño de código (hay un número máximo de bloques que pueden juntarse en el espacio de trabajo y el sistema no permite introducir ninguno nuevo si se llega a este número, hasta que se quite alguno). Se muestra una pista *"Las computadoras tienen memoria limitada. Llega al final de este camino usando tan solo dos bloques. Utiliza 'repetir' para ejecutar un bloque más de una vez"*. De este modo, los aprendices están obligados a usar el bloque de estructura repetitiva para resolver un problema de camino directo: `[rep [ad]]`. A partir de este momento la longitud del código está limitada, para que los aprendices deban resolver los niveles sin usar

las triviales, aunque laboriosas, soluciones secuenciales. Al mover las dos piezas se muestra un mensaje informativo *"Has usado todos los bloques para este nivel. Para añadir un bloque nuevo, primero debes eliminar un bloque existente"*.

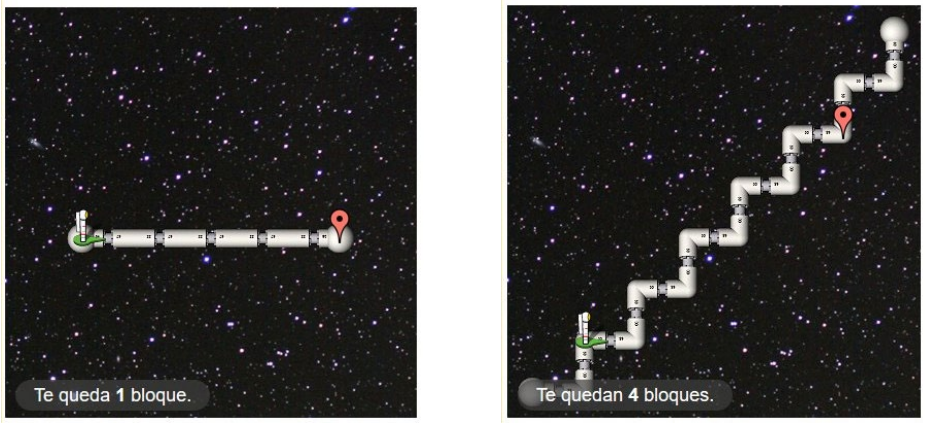


Figura 3.8. Laberintos de niveles 8 (izda.) y 9 (dcha.) de Kodetu.

El nivel 9 (figura 3.8, derecha) requiere repetición de una secuencia de cuatro pasos, con lo que el aprendiz debe identificar cuál es el patrón de pasos que hay que repetir. Se muestra una pista *"Puedes encajar más de un bloque dentro de un bloque 'repetir'"*. La única solución posible, al limitarse a 5 bloques, es `[rep [ad gi ad gd]]`.

El nivel 10 (figura 3.9, izquierda) incorpora la necesidad de una secuencia que no se repita antes del bucle, limitando con 5 bloques de nuevo a una sola solución posible: `[ad ad gi rep [ad]]`.

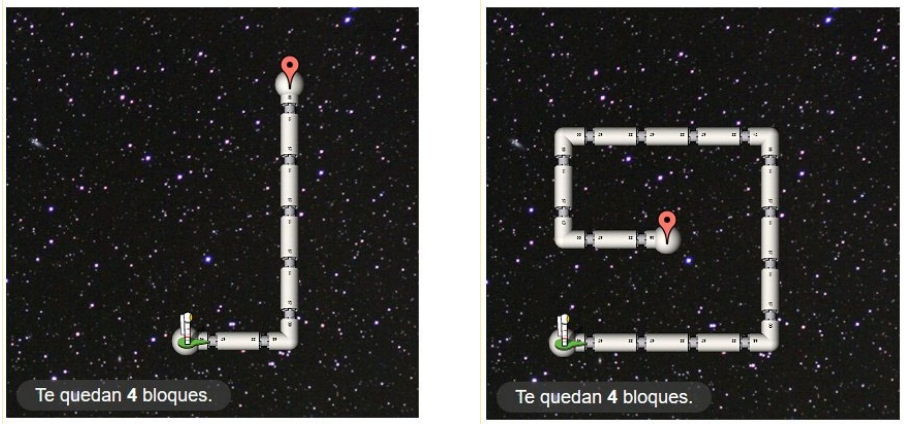


Figura 3.9. Laberintos de niveles 10 (izda.) y 11 (dcha.) de Kodetu.

El nivel 11 (figura 3.9, derecha) introduce los bloques de estructura condicional de una rama, donde se puede preguntar por la existencia o no de camino lateral o frontal para realizar una operación de giro solo en algunas ocasiones, dentro de la estructura repetitiva (instrucciones `si-i` si la condición es que haya camino a la izquierda, `si-d` a la derecha, `si-ad` adelante). Se muestra la pista *"Una condición 'si' va a hacer algo solamente si la condición es verdadera. Intenta girar a la izquierda si hay camino a la izquierda"*. Se mantiene el límite de 5 bloques, pero a

partir de este nivel se permite mayor expresividad de código, al existir varias soluciones posibles. La más eficiente en este caso es `[rep [ad si-i [gi]]]`.

El nivel 12 (figura 3.10, izquierda) mantiene el mismo límite y requiere cambiar la condición para adaptarla a una posibilidad diferente de giro (a la derecha en lugar de a la izquierda) en un laberinto con más distractores y varios caminos posibles. Se muestra la pista *"Haz clic en 'delante' en el bloque 'si' para cambiar su condición"*. Solución esperable: `[rep [ad si-d [gd]]]`.

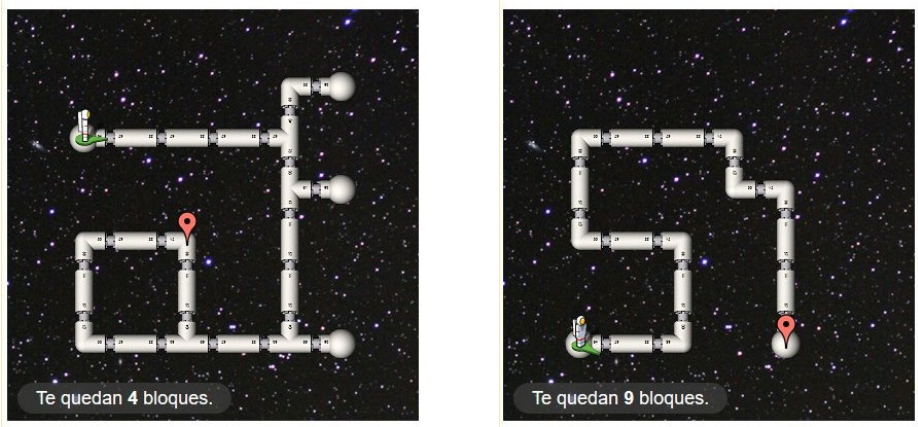


Figura 3.10. Laberintos de niveles 12 (izda.) y 13 (dcha.) de Kodetu.

En el nivel 13 (figura 3.10, derecha) introducimos el último bloque, la estructura condicional doble (**`si` / `si-no`**), que permite ejecutar una serie de bloques si existe camino en alguna de las tres direcciones y otra serie de bloques si no lo hay. Se muestra la pista *"Las declaraciones 'si-si no' hacen una cosa u otra dependiendo de si se cumple una condición o no"*. Este nivel necesita al menos tres estructuras de control (una repetición y dos condicionales) para resolver un laberinto que requiere giros en las dos direcciones y avances. Decidimos ampliar significativamente el límite (10 bloques) en este nivel para poder observar la expresividad de los aprendices cuando hay muchas soluciones posibles que resuelven el mismo laberinto. Una solución: `[rep [si-ad [ad] si-no [gi] si-d [gd]]]`.

El nivel 14 (figura 3.11, izquierda) es una prueba de conjunto que usa todos los conceptos previos para resolver un laberinto más complicado, aunque tiene un camino sencillo si se observa que puede hacerse solo con giros a izquierda. Límite de 7 bloques. Posible solución: `[rep [si-ad [ad] si-no [gi]]]`.

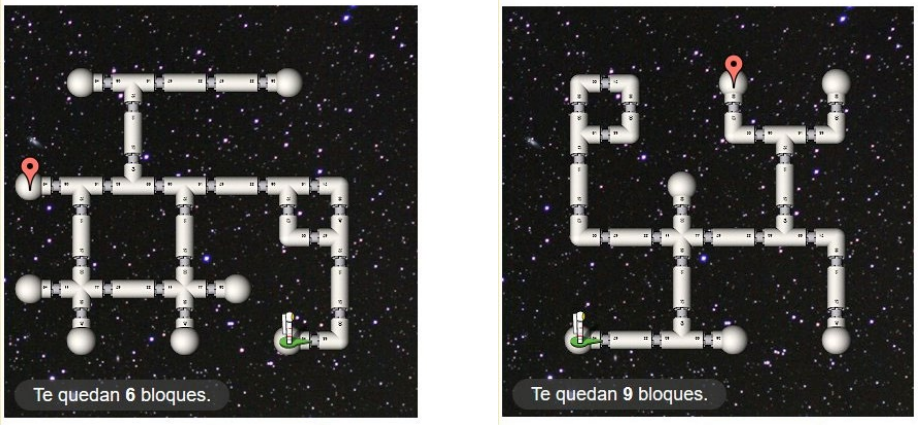


Figura 3.11. Laberintos de niveles 14 (izda.) y 15 (dcha.) de Kodetu.

Finalmente, el nivel 15 (figura 3.11, derecha) contempla el problema de un laberinto general con una significativa dificultad. Supone un reto hasta para programadores iniciados (requiere decidir cuál de las decisiones de giro o avance hay que consultar primero, porque de no hacerlo en el orden correcto el astronauta se mete en un ciclo inacabable). Hay varios caminos de distracción y una pista *"¿puedes resolver este complicado laberinto? Intenta seguir el truco de tocar siempre la pared de la izquierda"*. Límite de 10 bloques. Una solución posible: `[rep [si-i [gi ad] si-no [si-ad [ad] si-no [gd]]]]`. Para los aprendices que acaban este reto, se muestra una felicitación con el texto *"¡Felicidades! ¡Has resuelto el último nivel!"*.

El reto completo de los 15 niveles está diseñado para poderse realizar en una sesión de 40-60 minutos para aprendices de programación sin conocimientos previos, en 20-30 minutos para aprendices con conocimientos de programación o que hayan jugado previamente.

Adicionalmente, uno de los datos que nos pareció importante analizar es cómo influye en la evolución de un aprendiz su capacidad de elección del siguiente reto a afrontar. En particular, decidimos en Kodetu que una opción fuera que cada aprendiz esté obligado a resolver un nivel para poder afrontar el siguiente (como es habitual en muchos juegos y actividades de aprendizaje); y la otra opción que el aprendiz pueda libremente y en cualquier momento elegir el nivel que desee del 1 al 15, independientemente de su resultado o paso por los niveles anteriores.

De este modo, planteamos **dos modalidades** de Kodetu, una de libre navegación (figura 3.12) que muestra los niveles disponibles y permite moverse por ellos aunque el sistema sugiere el tránsito secuencial (que es el que ocurre si no se utiliza la navegación directa); y otra que, mostrando el nivel, obliga a realizarlos de forma estrictamente secuencial y exitosa (no se presenta un nivel hasta que no se haya solucionado el anterior).

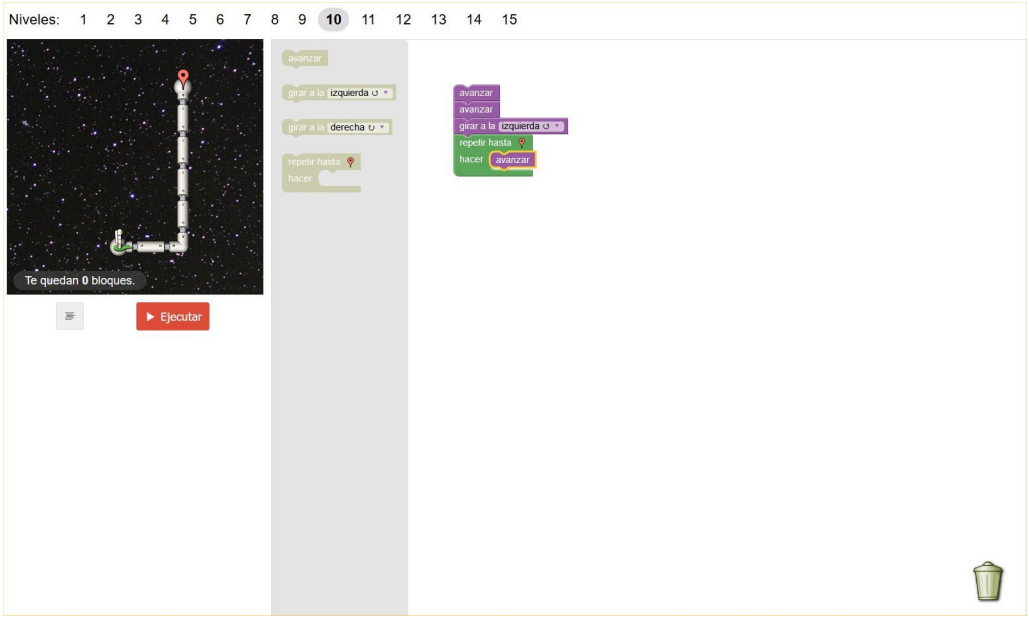


Figura 3.12. Interfaz de Kodetu, versión navegación libre.

La herramienta Kodetu está disponible en la web pública para cualquier aprendiz desde 2014. Por ello hay muchos aprendices no controlados, anónimos, que han jugado e intentado resolver los distintos niveles. Además, hemos organizado desde octubre de 2014 grupos supervisados de aprendices jugando a la vez, con un ordenador por persona y de forma individual, en sesiones con una duración aproximada de 50 minutos. El nivel al que llega cada aprendiz puede verse como un indicador inicial de su habilidad para aprender conceptos de programación, aunque como desarrollaremos más adelante hay muchos más indicadores que podemos tener en consideración para el estudio.

3.2.2. Simulación de sesión: KodetuWin

Para analizar los registros de los aprendices a lo largo de su experiencia en Kodetu, necesitamos observar datos específicos de cada aprendiz en sus interacciones dentro de cada nivel, que describen pormenorizadamente cómo el código va evolucionando. Dado que la observación ocurre siempre a posteriori de la experiencia del aprendiz, para poder realizar este proceso desarrollamos una herramienta software de soporte a la que hemos llamado *KodetuWin*. Desarrollada en Java, esta utilidad funciona como un navegador especializado en el registro de datos de Kodetu (figura 3.13), permitiéndonos acceder a los datos de cada sesión de aprendiz (panel superior), nivel, e interacción dentro del nivel (panel central). Seleccionando cada interacción, KodetuWin permite observar (panel inferior) la representación (textual y anidada) del código del aprendiz en ese punto, el laberinto correspondiente y el camino que ese código sigue en el laberinto. En

ocasiones, es importante observar los pasos intermedios, tanto en el código como en el camino que ese código genera, incluso si el aprendiz no ejecutó el código.

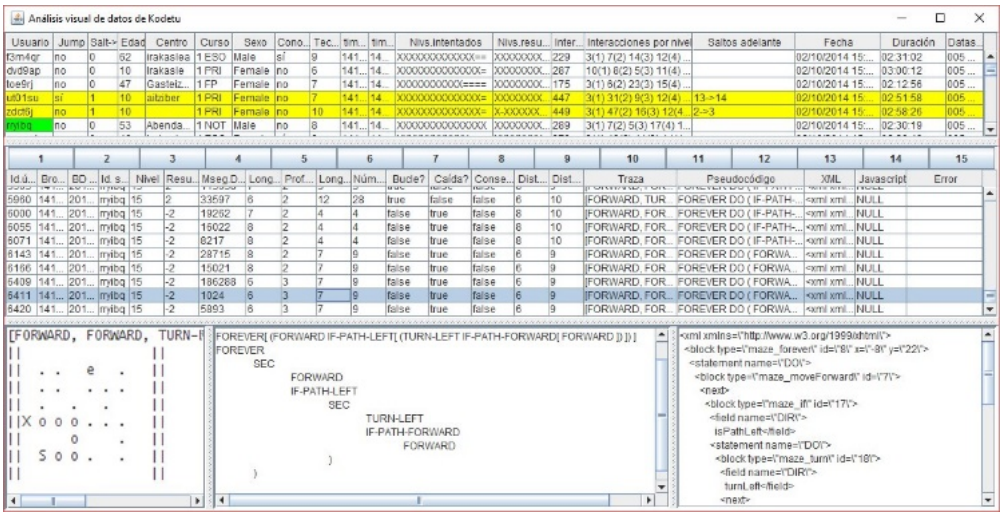


Figura 3.13. Vista del interfaz principal de KodetuWin.

Esta herramienta permite analizar cuándo los códigos resuelven o no el nivel y las diferencias en código entre los intentos y las soluciones óptimas a cada nivel. Pero también nos ha parecido importante comprobar cómo de lejos está de la solución en términos de código el resultado del código propuesto por cada aprendiz. Por ello, incorporamos a KodetuWin un simulador de ejecución que permite tomar cada código de aprendiz y realizar una simulación de su ejecución sobre el laberinto de nivel correspondiente. Así, podemos calcular el camino del astronauta en cualquier momento si el aprendiz hubiera intentado ejecutar ese código. Este software nos ha permitido obtener información profunda del proceso completo de cada aprendiz, como soluciones no intentadas (configuraciones de código que habrían resuelto el reto pero que el aprendiz no intentó reproducir -no pulsó el botón *ejecutar*-) o indicadores que miden la proximidad de una solución al objetivo esperado, desde el punto de vista de la construcción del código y desde el punto de vista del camino real seguido por el astronauta al ejecutar el código construido. Además del registro original de cada interacción, la información adicional que hemos generado para su posible utilidad en la investigación incluye los siguientes datos:

- Tiempo transcurrido en cada interacción desde la interacción anterior.
- Diferencia de código: número de bloques eliminados, modificados o añadidos desde la interacción anterior.
- Longitud de código: número de bloques de código en cada interacción.
- Profundidad de código: número mayor de anidamiento de estructuras de control en esa interacción.
- Información de éxito de ese código (correcto, no llega a meta, bucle, caída).
- Longitud del camino que ese código realiza en el laberinto.
- Número de pasos que ese código ejecuta en el laberinto (giros o avances).

- Número de bloques de código muerto: bloques que no se llegan a ejecutar, porque la ejecución acaba o entra en bucle antes de llegar a ellos.
- Coste de ejecución de ese código en el laberinto: número de bloques que se ejecutan, incluyendo condiciones que se comprueban en alternativas.
- Distancia de camino a la meta: número de pasos que quedarían por dar desde el camino realizado para alcanzar la meta.
- Traza del camino realizado.
- Traza del código ejecutado.

Partiendo de estos datos, se generan indicadores acumulados por nivel y aprendiz:

- Número de interacciones por nivel.
- Tiempo de nivel: tiempo consecutivo que usa cada aprendiz en un nivel, desde que lo inicia hasta que lo resuelve, finaliza o cambia de nivel.
- Intentos por nivel: número de veces que el aprendiz pulsa el botón de ejecución, intentando resolver ese nivel.
- Diferencia de código por nivel: número de bloques eliminados, modificados y añadidos.
- Código de solución de nivel (si existe), con todos sus valores asociados (longitud, profundidad, longitud de camino, número de bloques ejecutados, bloques de código muerto, coste de ejecución, distancia a la meta, traza de camino, traza de código).

3.2.3. Conjunto de datos

En la primera fase de nuestro estudio, después de eliminar datos de prueba y aprendices incorrectos (sesiones sin interacciones), nuestro primer conjunto de datos contiene 3.355 sesiones de juego, con 1.097.261 interacciones. Estos datos fueron registrados entre octubre de 2014 y marzo de 2016.

De cara al análisis, dividimos los datos en tres grupos:

- Grupo A: aprendices de programación en grupos supervisados utilizando la versión estándar de Kodetu (los niveles deben resolverse obligatoriamente de forma secuencial).
- Grupo B: aprendices de programación en grupos supervisados utilizando la versión de navegación libre de Kodetu (los niveles pueden ser completados en cualquier orden).
- Grupo C: resto de aprendices (aprendices no supervisados, de cualquier edad, en laboratorio o en lugar desconocido a través de Internet), tanto en la versión libre como la secuencial. La elección de versión no se dejó de forma explícita a los aprendices: algunos se encontraron con la versión secuencial y otros se encontraron con la versión libre.

En nuestro conjunto de datos, el grupo A está formado por 916 sesiones realizadas por aprendices que se organizaron en 37 grupos supervisados, que completaron 384.213 interacciones. En este grupo, 415 (45,3%) de las sesiones fueron realizadas por chicas y 501 por chicos y el 63,9% son aprendices que manifestaron no tener ninguna experiencia de programación previa. La edad media de los aprendices es 11,5 años (DT: 2,17), el 93,8% de los mismos está entre 9 y 17 años y la afinidad con la tecnología media (en un rango de 1 a 10) es de 8,48 (DT: 2,0).

El grupo B incluye 1.115 aprendices divididos en 32 grupos supervisados, completando 358.202 interacciones. 504 (el 45,2%) de estas sesiones fueron realizadas por chicas y 611 por chicos y el 67,4% son aprendices que manifiestan no tener experiencia previa de programación. La edad media del aprendiz es 11,2 años (DT: 4,2) y la afinidad tecnológica media es de 8,63 (DT: 1,8).

El grupo C incluye 1.324 aprendices y 354.846 interacciones. En este grupo, 578 (el 43,7%) de las sesiones fueron realizadas por mujeres/niñas y 745 por hombres/niños. El 63,0% de los aprendices indican no tener experiencia previa de programación. La edad media es 15,0 años (DT: 9,1), el 68,8% tienen entre 9 y 17 años y la afinidad tecnológica es de 7,79 (DT 2,5).

3.3. Resultados

En esta sección, analizamos los resultados específicos de los aprendices de este conjunto de datos que utilizaron Kodetu, diferenciando sus grupos.

3.3.1. Influencia de las características de los aprendices

Relación con la edad

En muchos juegos y actividades de aprendizaje, la organización de los contenidos por niveles de habilidad o conocimiento es un aspecto fundamental. La dificultad ajustada a la capacidad del aprendiz y el incremento progresivo de esta dificultad facilitan el aprendizaje y mantiene el interés. Con Kodetu, debido al tiempo limitado que hemos planteado en los grupos supervisados, solo el 10,5% de los aprendices en los grupos A y B consiguieron completar el nivel 15.

La tabla 3.1 muestra cómo el porcentaje de aprendices que completa cada nivel va decreciendo progresivamente. El modo libre que permite a los aprendices elegir el nivel intentado (grupo B) afecta al porcentaje de aprendices que completan los niveles: en el grupo A (juego secuencial) solo el 7,6% de los aprendices completa todos los niveles, pero en el grupo B (juego libre) lo hace el 10,8%.

GRUPO A	Nivel														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	100,0%	93,5%	93,5%	90,3%	90,3%	83,9%	71,0%	67,7%	67,7%	54,8%	29,0%	16,1%	3,2%	0,0%	0,0%
10	100,0%	97,7%	95,4%	94,6%	91,9%	88,0%	84,6%	81,9%	73,4%	66,8%	45,2%	32,0%	19,3%	8,5%	2,3%
11	99,7%	98,0%	95,0%	93,3%	91,7%	91,0%	88,7%	87,3%	85,7%	80,0%	71,3%	50,3%	22,3%	12,7%	1,3%
12	100,0%	99,1%	95,6%	93,0%	93,0%	91,2%	90,4%	89,5%	89,5%	86,0%	76,3%	62,3%	25,4%	15,8%	6,1%
13	100,0%	96,9%	96,9%	95,3%	93,8%	92,2%	89,1%	89,1%	87,5%	84,4%	75,0%	73,4%	62,5%	54,7%	21,9%
14	100,0%	100,0%	100,0%	100,0%	96,6%	89,7%	89,7%	89,7%	86,2%	86,2%	79,3%	69,0%	44,8%	31,0%	13,8%
15	100,0%	87,5%	87,5%	87,5%	87,5%	81,3%	81,3%	81,3%	81,3%	75,0%	68,8%	62,5%	43,8%	43,8%	37,5%
16	100,0%	100,0%	96,0%	96,0%	96,0%	96,0%	96,0%	96,0%	96,0%	92,0%	88,0%	84,0%	80,0%	72,0%	52,0%
17	100,0%	95,2%	90,5%	90,5%	90,5%	85,7%	85,7%	85,7%	85,7%	81,0%	71,4%	71,4%	71,4%	66,7%	38,1%
Resto	96,5%	94,7%	94,7%	91,2%	80,7%	71,9%	61,4%	57,9%	49,1%	47,4%	38,6%	29,8%	22,8%	21,1%	14,0%
Total	99,7%	97,5%	95,2%	93,7%	91,5%	88,6%	85,5%	83,8%	80,1%	74,9%	62,0%	48,0%	27,8%	18,9%	7,6%

GRUPO B	Nivel														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	95,8%	89,5%	87,4%	86,3%	84,2%	82,1%	60,0%	53,7%	47,4%	37,9%	29,5%	13,7%	3,2%	3,2%	0,0%
10	87,6%	86,2%	82,8%	80,1%	76,9%	73,3%	69,4%	67,5%	64,8%	58,7%	41,7%	27,2%	14,3%	11,4%	6,3%
11	88,2%	84,9%	81,8%	80,0%	76,4%	72,9%	68,9%	67,3%	63,6%	60,7%	51,3%	40,9%	26,7%	20,9%	12,4%
12	94,3%	90,6%	84,9%	84,9%	83,0%	79,2%	77,4%	77,4%	75,5%	66,0%	41,5%	41,5%	28,3%	18,9%	32,1%
13	96,2%	88,5%	84,6%	84,6%	84,6%	84,6%	80,8%	76,9%	76,9%	80,8%	69,2%	73,1%	53,8%	53,8%	38,5%
14	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	66,7%	66,7%	66,7%	66,7%
15	80,0%	75,0%	70,0%	70,0%	60,0%	75,0%	80,0%	65,0%	70,0%	70,0%	75,0%	70,0%	65,0%	70,0%	20,0%
16	93,5%	90,3%	90,3%	83,9%	87,1%	83,9%	80,6%	80,6%	83,9%	83,9%	87,1%	87,1%	77,4%	61,3%	6,5%
17	66,7%	66,7%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	66,7%	66,7%	66,7%	33,3%	33,3%	33,3%	0,0%
Resto	86,4%	77,3%	77,3%	72,7%	72,7%	68,2%	59,1%	68,2%	63,6%	59,1%	50,0%	50,0%	45,5%	50,0%	13,6%
Total	89,1%	85,9%	82,9%	80,8%	77,8%	74,8%	69,5%	67,4%	64,3%	59,6%	47,4%	36,3%	23,4%	19,3%	10,8%

GRUPO C	Nivel														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	92,7%	90,9%	80,0%	74,5%	70,9%	60,0%	52,7%	47,3%	40,0%	29,1%	27,3%	21,8%	12,7%	12,7%	1,8%
10	82,1%	83,2%	73,2%	67,5%	61,8%	58,6%	53,9%	48,2%	46,4%	41,4%	30,4%	25,4%	18,6%	14,6%	4,6%
11	84,3%	81,7%	76,2%	72,3%	70,6%	64,3%	57,9%	56,2%	54,9%	51,5%	43,0%	36,6%	23,0%	20,9%	4,3%
12	87,9%	86,8%	81,3%	76,9%	75,8%	68,1%	64,8%	61,5%	59,3%	53,8%	44,0%	33,0%	24,2%	22,0%	9,9%
13	94,0%	91,0%	89,6%	86,6%	83,6%	82,1%	77,6%	74,6%	74,6%	70,1%	64,2%	61,2%	49,3%	41,8%	23,9%
14	98,7%	98,7%	94,7%	92,0%	88,0%	84,0%	84,0%	81,3%	84,0%	81,3%	74,7%	68,0%	53,3%	48,0%	18,7%
15	97,3%	97,3%	89,2%	89,2%	78,4%	73,0%	64,9%	62,2%	48,6%	45,9%	35,1%	35,1%	29,7%	24,3%	5,4%
16	96,6%	94,9%	93,2%	93,2%	88,1%	88,1%	81,4%	83,1%	78,0%	72,9%	67,8%	61,0%	54,2%	49,2%	16,9%
17	91,7%	91,7%	91,7%	91,7%	75,0%	66,7%	58,3%	66,7%	58,3%	50,0%	50,0%	50,0%	41,7%	33,3%	8,3%
Resto	93,7%	91,3%	86,2%	81,8%	79,7%	76,8%	71,9%	71,7%	70,0%	68,0%	63,7%	59,1%	50,1%	44,3%	20,6%
Total	89,7%	88,3%	82,2%	78,1%	74,6%	70,4%	65,4%	63,1%	61,0%	57,2%	50,0%	44,6%	35,0%	30,7%	12,2%

Tabla 3.1. Porcentaje de aprendices que completan cada nivel (columna) con respecto a los que inician Kodetu, por edad (fila). Grupo A=Juego secuencial, grupo B=Juego libre, grupo C=no supervisados.

La edad, como es lógico esperar y así aparece en la literatura (Román-González, Pérez-González y Jiménez-Fernandez, 2017), mejora la capacidad de finalizar todos los niveles, pero en Kodetu solo hasta un cierto punto (13-14 años). A partir de ese punto, la mejora no es significativa e incluso decae en edades adultas. El test de correlación de Spearman muestra que se encuentra una correlación positiva entre estas variables en todos los grupos (grupo A: r_s=0,295, p<0,001; grupo B: r_s=0,356, p<0,001; grupo C: r_s=0,255, p<0,001). Sin embargo, los aprendices de 15 a 17 años no se comportan mejor que los de 13 y 14 años.

También encontramos que los aprendices que resolvieron más retos necesitaron menos tiempo medio (grupo A: r_s=0,278, p<0,001; grupo B: r_s=0,236, p<0,001; grupo C: r_s=0,213, p<0,001). La figura 3.14 muestra esta relación entre edad y tiempo de solución, donde se puede observar el tiempo medio por nivel y edad.

Los cambios sustanciales se observan solo en el nivel 15, pero este nivel no es estadísticamente significativo al ser muy pequeño el número de aprendices que finalizan el último nivel en algunas edades (p. ej. N=21 para personas de 16 años, N=8 para 17).

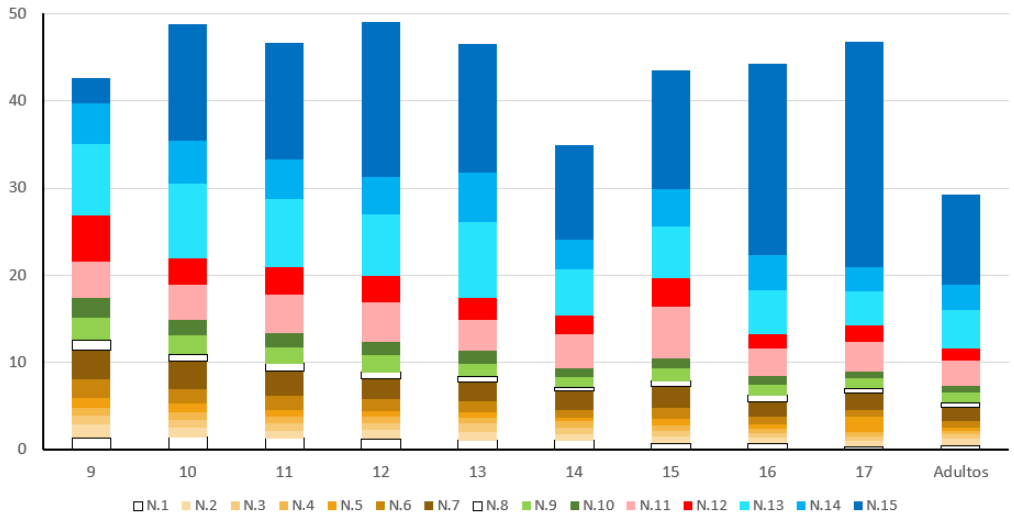


Figura 3.14. Tiempo medio apilado en niveles resueltos (minutos, eje Y) por edad (eje X). Niveles con diferente coloreado desde abajo (nivel 1) hacia arriba (nivel 15).

Relación con el sexo

Se encuentran diferencias significativas entre sexos (tabla 3.2). En todos los niveles y en todos los grupos, el porcentaje de chicos que completa un nivel determinado es mayor que el de chicas. La diferencia es mayor en los niveles 11 y 12 y disminuye en los niveles superiores. Un número similar de chicos y chicas finaliza los retos de Kodetu, pero las chicas tienden a abandonar el juego antes que los chicos. Esta tendencia se observa en todas las edades.

GRUPO A	Nivel														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Chicos	99,8%	98,2%	96,0%	94,8%	92,6%	90,4%	87,6%	85,8%	82,6%	76,0%	65,9%	52,9%	31,3%	22,0%	9,4%
Chicas	99,5%	96,6%	94,2%	92,3%	90,1%	86,5%	82,9%	81,4%	77,1%	73,5%	57,3%	42,2%	23,6%	15,2%	5,5%
Total	99,7%	97,5%	95,2%	93,7%	91,5%	88,6%	85,5%	83,8%	80,1%	74,9%	62,0%	48,0%	27,8%	18,9%	7,6%

GRUPO B	Nivel														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Chicos	89,2%	86,3%	84,6%	83,1%	80,5%	77,3%	73,8%	72,5%	69,9%	65,3%	55,0%	42,6%	25,4%	20,6%	11,0%
Chicas	88,9%	85,5%	80,8%	78,0%	74,6%	71,8%	64,3%	61,3%	57,5%	52,8%	38,3%	28,8%	21,0%	17,7%	10,5%
Total	89,1%	85,9%	82,9%	80,8%	77,8%	74,8%	69,5%	67,4%	64,3%	59,6%	47,4%	36,3%	23,4%	19,3%	10,8%

GRUPO C	Nivel														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Chicos	90,7%	89,5%	84,6%	80,7%	78,0%	74,0%	69,0%	66,3%	65,0%	60,9%	55,2%	47,9%	37,4%	32,8%	12,8%
Chicas	88,3%	86,7%	79,1%	74,8%	70,3%	65,8%	60,8%	59,1%	56,0%	52,3%	43,4%	40,2%	31,8%	28,0%	11,4%
Total	89,7%	88,3%	82,2%	78,1%	74,6%	70,4%	65,4%	63,1%	61,0%	57,2%	50,0%	44,6%	35,0%	30,7%	12,2%

Tabla 3.2. Porcentaje de aprendices que acaban cada nivel, por sexo y grupo.

El test de Mann-Whitney indica que el nivel mayor resuelto por cada aprendiz es significativamente mayor en chicos que en chicas en todos los grupos (grupo A: U=107364, p<0,001; grupo B: U=78869,5, p=0,001; grupo C: U=74644,5, p<0,001). También encontramos que ambos sexos usan su tiempo de forma diferente. Los chicos son más rápidos resolviendo los retos en los primeros niveles (1 a 7), donde las habilidades predominantes son la repetición y la orientación espacial. De media, los chicos resuelven 6,9 segundos más rápido el nivel 7 que las chicas. Esta tendencia se invierte cuando aparecen las estructuras de control (niveles 9 a 15) y las chicas son significativamente más rápidas resolviendo los dos últimos niveles (3,3 y 27,8 segundos de diferencia en los niveles 14 y 15). En general, el tiempo invertido en resolver cada nivel es significativamente mayor para chicos que para chicas en los grupos A y C (M-W para el grupo A: U=119375, p<0,001; grupo C: U=81589,5, p=0,002), pero no para el grupo B.

El número de intentos (ejecuciones) que hace un aprendiz en cada nivel es uno de los criterios que está relacionado con la capacidad de programación. Cuando un aprendiz hace *n* intentos en un nivel que finalmente se resuelve, esto puede interpretarse como que se han cometido *n-1* errores y *1* acierto. En general, los aprendices masculinos hacen menos intentos que los femeninos, pero la diferencia es pequeña (0,1 de media). La tabla 3.3 muestra el detalle por edad y nivel (celdas de color cian, mejor comportamiento de los chicos, celdas de color magenta, mejor comportamiento de las chicas). Las chicas tienden a comportarse mejor en los niveles superiores (11 a 14).

En los grupos A y C, los chicos hicieron más intentos que las chicas (M-W: grupo A: U=129578,5, p<0,001; grupo C: U=88781, p=0,001).

Nivel / Edad	Número de nivel															Prom.
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
9	0,1	0,1	-0,1	-0,4	-0,4	-0,9	-0,6	0,2	0,8	0,2	0,6	-1,7	-2,1	2,3	-1,9	-0,26
10	0,0	0,1	-0,2	-0,4	-0,2	-0,5	-0,6	0,0	-0,2	0,2	0,3	-0,2	1,2	1,2	-1,0	-0,02
11	0,1	-0,1	-0,2	-0,5	-0,4	-0,6	-0,5	-0,1	-0,2	-0,1	-0,1	0,0	1,0	1,1	0,5	0,00
12	0,1	0,0	-0,2	-0,5	-0,4	-0,8	-0,5	-0,8	-0,8	0,3	-1,1	1,0	1,3	-1,8	0,6	-0,24
13	0,1	-1,2	-1,4	-0,9	-1,1	-1,8	-1,5	-0,3	0,6	-0,1	1,4	0,3	-2,3	-1,0	5,4	-0,26
14	0,3	-0,2	0,1	1,0	0,0	0,1	1,1	0,4	1,2	0,6	1,5	0,4	-3,4	0,6	-12,8	-0,61
15	0,2	-0,9	0,0	0,2	-0,3	-0,3	-0,9	0,2	-4,1	-1,7	-0,1	-1,3	-0,8	2,1	-1,2	-0,60
16	-0,1	0,0	-0,3	-0,5	-0,8	-0,2	-1,0	0,4	-0,4	-0,3	-1,6	-0,1	0,1	-2,1	0,3	-0,45
17	-0,3	-0,1	-0,8	-0,5	-1,5	-1,0	-1,8	-1,2	-1,6	-3,0	-5,5	-0,3	-2,5	-3,0	-2,4	-1,69
Resto	-0,2	0,0	-0,1	-0,2	-0,5	-1,1	-0,4	0,1	-0,6	0,3	1,3	-1,9	2,1	1,4	9,3	0,64
Total	0,0	-0,1	-0,2	-0,3	-0,4	-0,6	-0,5	-0,1	-0,2	0,0	0,1	-0,2	0,2	0,3	-0,2	-0,14

Tabla 3.3. Diferencia de intentos entre chicos y chicas por nivel
(cian: mejor los chicos; magenta: mejor las chicas).

Otro factor que podemos analizar en el proceso de solución de Kodetu es el número de cambios que cada aprendiz hace en el código mientras intenta resolver el reto. Analizamos la evolución de código de cada aprendiz observando tres elementos: cuántos bloques añade (los que lleva de la zona de bloques a la zona de trabajo), cuántos borra (los que quita del espacio de trabajo) y cuántos modifica (los

que cambia permaneciendo en el espacio de trabajo: este caso se refiere a las estructuras alternativas, cuya condición puede modificarse sin cambiar el bloque). En los casos de añadido o borrado, en Kodetu pueden arrastrarse bloques combinados en un solo arrastre de ratón: en ese caso contamos todos los bloques individuales incluidos en el movimiento. Contando y totalizando el número de cambios realizados desde que empieza el nivel hasta que se acaba, encontramos un indicador de rendimiento.

Para todos los niveles hemos determinado el número óptimo de cambios (los que llevaría solo añadir bloques desde el código inicial hasta la solución más plausible) y calculamos la diferencia con respecto a ese número (operaciones adicionales necesitadas para completar el nivel). La tabla 3.4 muestra cómo este indicador se relaciona con las variables de edad y de sexo. Los chicos se comportan mejor en los niveles inferiores (por ejemplo, 2,5 cambios de código menos en el nivel 7) y las chicas se comportan mejor en los niveles superiores (por ejemplo, 9,2 cambios menos en el nivel 15).

Nivel / Edad	Número de nivel															Prom.
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
9	-0,1	-1,2	0,6	-1,9	1,6	2,0	-3,4	-1,1	-8,5	-2,8	-2,9	-3,8	44,5	-22,0		0,07
10	0,0	-0,1	-1,0	-0,7	-1,5	-1,1	-2,4	-0,7	-0,4	-3,8	0,5	2,4	-10,6	-1,3	17,4	-0,22
11	0,1	-1,2	-1,3	-1,1	-1,6	-2,6	-18,8	-1,2	-3,4	-3,1	1,4	-5,4	-3,7	2,1	11,5	-1,89
12	0,0	-1,5	-0,7	-0,2	-0,9	-1,1	0,4	-1,0	-6,4	0,2	-21,6	-10,3	-14,6	-11,2	-34,2	-6,87
13	0,0	-2,7	-4,2	-2,9	-3,4	-2,2	-5,7	-2,4	-4,5	-5,0	8,8	-0,3	-23,6	-3,1	24,6	-1,77
14	0,0	-1,6	-2,4	-2,2	-0,7	0,3	-0,3	0,1	0,5	-1,7	4,8	1,6	-1,3	14,1	14,0	1,68
15	0,0	-1,2	-2,1	-1,2	0,3	-2,6	1,3	-1,1	1,2	2,2	5,3	3,6	-13,6	-15,4	-3,4	-1,78
16	-0,1	-1,3	-1,1	-0,4	-0,8	0,0	-1,6	-2,3	-1,8	-3,6	-5,0	-0,7	-10,3	2,7	-31,3	-3,84
17	0,0	1,4	-1,4	0,3	-2,3	2,1	-5,5	0,8	-2,7	-1,0	23,0	-3,3	-1,1	19,1	-51,1	-1,45
Resto	0,2	-1,0	-1,8	-0,4	-0,5	0,0	-0,4	-1,6	-4,5	-1,5	-8,9	-3,2	-23,5	-6,1	-4,0	-3,81
Total	0,1	-1,0	-1,3	-0,9	-1,2	-1,1	-1,6	-1,1	-2,8	-2,2	-1,5	-2,1	-10,0	-0,7	2,2	-1,68

Tabla 3.4. Diferencia de cambios de código entre chicos y chicas por nivel
(cian: mejor los chicos; magenta: mejor las chicas).

Relación con el conocimiento previo de programación

Contrariamente a lo esperable, el conocimiento previo de programación no solo no correlaciona con el resultado, sino que influye negativamente en términos del porcentaje de nivel completado. En todas las edades y niveles un porcentaje mayor de aprendices consigue superar el nivel si el aprendiz manifiesta no tener conocimiento previo de programación. En general, hay un 64,2% de éxitos entre los aprendices sin conocimientos previos de programación y un 59,7% entre los aprendices con conocimientos previos. No obstante, en los niveles superiores el porcentaje se acerca (10,8% y 9,8%, respectivamente, en el nivel 15). En los grupos supervisados las diferencias en el nivel 15 son especialmente bajas (0,1% en el grupo A y 0,4% en el grupo B), pero la tendencia se mantiene.

De este modo, los aprendices con conocimientos previos de programación resolvieron menos retos que aquellos sin conocimientos previos, en los grupos B y

C (M-W para el grupo B: U=74514,5, p=0,008; grupo C: U=78078, p=0,008); no se encontraron diferencias en el grupo A.

Los conocimientos previos sí afectan positivamente al tiempo de solución, particularmente cuando la complejidad aumenta. Considerando todos los retos, los aprendices con conocimientos previos fueron significativamente más rápidos en resolverlos (M-W para el grupo A: U=112384, p=0,085; grupo B: U=66502, p<0,001; grupo C: U=80405, p=0,001), específicamente desde el nivel 7 (3,2 segundos más rápidos de media) hasta el reto 13 (23.8 segundos más rápidos de media).

Sin embargo, en los niveles superiores 14 y 15, donde se necesita la abstracción para encontrar la solución general a un laberinto, el conocimiento previo no conlleva una ventaja significativa. En estos niveles se invierte la tendencia y los aprendices sin experiencia previa encuentran la solución 11,1 segundos más rápido.

Al respecto de los cambios de código, los aprendices con experiencia previa de programación requirieron menos cambios para encontrar la solución en todos los niveles, excepto en el nivel 15 donde esta tendencia de nuevo se invierte (los aprendices sin conocimientos previos realizaron 2,2 cambios menos de media).

Relación con la afinidad tecnológica

En los resultados con respecto a la afinidad con la tecnología también aparecen algunas tendencias no esperables. No se ha podido encontrar correlación entre el rendimiento y el gusto por la tecnología. Sin embargo, los aprendices con los valores mayores de afinidad tecnológica (10) se comportan significativamente peor que aquellos con valores altos (7, 8 o 9) en todos los grupos del conjunto de datos. El éxito en el nivel 15 es especialmente interesante, porque los aprendices con valores bajos de afinidad tecnológica (1 y 2) son los más exitosos llegando y resolviendo este nivel, especialmente si participan con juego secuencial (25% de los que juegan secuencialmente, 4,8% de los que lo hacen con saltos). En los niveles 11 a 15, los más relacionados con la habilidad algorítmica, encontramos buen comportamiento entre los aprendices con valores altos de afinidad (7 a 9), pero no entre los que tienen el valor máximo (10) (figura 3.15).

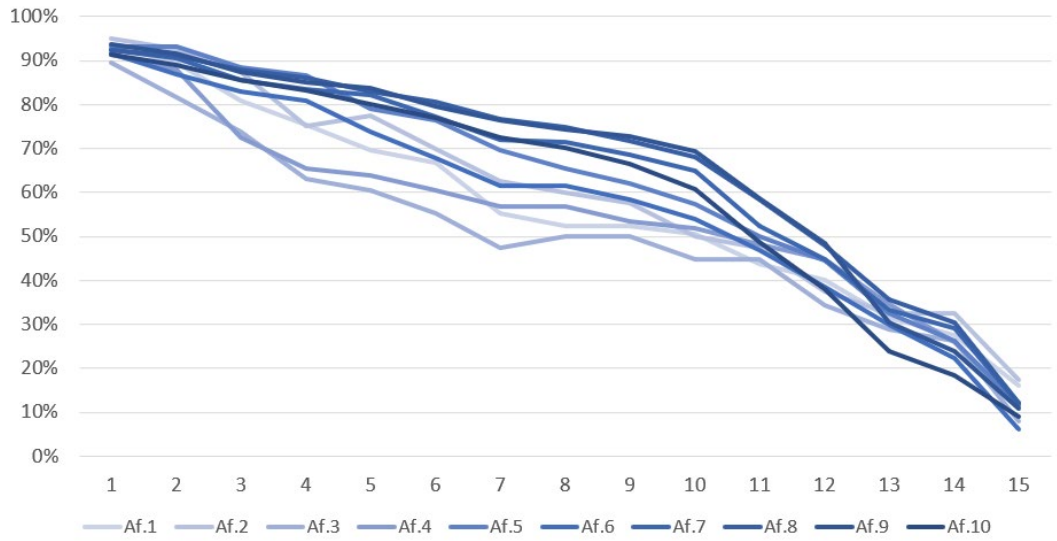


Figura 3.15. Relación entre afinidad con la tecnología (color) y porcentaje de aprendices (eje Y) que superan cada nivel (eje X).

El coeficiente de Spearman muestra una correlación positiva entre el rendimiento y la afinidad tecnológica en el grupo A (r_s=0,126, p<0,001), una correlación negativa en el grupo B (r_s=-0,128, p<0,001) y ausencia de correlación en el grupo C.

No hay relación entre la afinidad tecnológica y el tiempo de solución o el número total de intentos. Los aprendices que puntúan 10 no son los más rápidos en ningún nivel y, curiosamente, los aprendices con puntuaciones más bajas (1-4) son más rápidos en la mayoría de los niveles.

3.3.2. Análisis de código

Para analizar cuántas soluciones diferentes encuentran los aprendices ante los mismos problemas, observamos los códigos propuestos por los aprendices como soluciones válidas de los diferentes niveles. En el PC es esencial considerar que puede haber varias soluciones válidas ante un mismo problema. Analizamos todos los códigos exitosos y encontramos una particular configuración de soluciones: un gran porcentaje de aprendices eligen unas pocas soluciones comunes y el resto de soluciones se extienden en muchas soluciones distintas con distintas variaciones, a modo de "largas colas", como se ve en la figura 3.16. A pesar del contexto limitado que supone Kodetu para la expresividad del código propuesto como solución (en algunos niveles esta expresividad es imposible), encontramos varios niveles con información relevante.

Figura 3.16. Largas colas de variantes de código (h.) por cada nivel (v.). Cada línea marca el porcentaje de aprendices que repiten cada código único de solución, de izquierda (códigos más comunes) a (códigos menos comunes). Líneas más largas indican más códigos diferentes.

Para exponer códigos de solución, recordamos las abreviaturas para los distintos bloques que se utilizan en Kodetu:

- `ad` - Movimiento hacia adelante
- `gi` - Girar a la izquierda
- `gd` - Girar a la derecha
- `si-ad` - Condicional si hay camino adelante
- `si-i` - Condicional si hay camino a la izquierda
- `si-d` - Condicional si hay camino a la derecha
- `si-no` - Condicional de dos ramas (inicia la segunda parte de la estructura condicional)
- `rep` - Repetir bloques hasta que se encuentra la meta (bucle)

El 85,5% de las soluciones al nivel 1 fueron la esperada `[ad ad]`. Con una solución tan simple para un nivel de introducción, es posible que probar el sistema o malinterpretar el botón de ejecución causara a tantos aprendices (14,5%) encontrar otras soluciones. La mayor parte utilizan código muerto (añadir bloques que ya no se ejecutan porque se ha llegado a la meta, como `[ad ad ad]`). Otros incluyen código nulo, por ejemplo un giro a la izquierda seguido de un giro a la derecha, que se anulan mutuamente (`[ad gi gd ad]`).

En los niveles 2 a 6, a pesar de ser secuenciales y tener caminos relativamente cortos, se han encontrado también muchas variantes. Entre el 58% y el 95,5% de las soluciones son la esperada. Las no esperadas, además de código muerto, incorporan variantes de giros laterales (`[gi gi gi]` en lugar de `[gd]`, por ejemplo). En el nivel 4, se encuentran hasta 38 soluciones diferentes además del

código óptimo de cinco pasos: `[gi gi ad ad ad]`. En el nivel 7, donde varios caminos posibles de distintas eficiencias pueden superar el reto, encontramos hasta 65 soluciones alternativas (la menos eficiente utiliza hasta 27 unidades de código más que las mínimas necesarias).

Los niveles 8, 9 y 10 son excepciones en términos de código. Al ser utilizadas para el aprendizaje de estructuras repetitivas e incorporar límites muy estrictos de longitud de código, fuerzan una única solución posible. Desde aquí la limitación se relaja, lo que permite encontrar expresividad y riqueza de código a partir del nivel 11.

En los niveles 11 y 12, aunque solo pueden usarse 4 y 5 bloques, se encuentran un significativo número de variantes. En el nivel 11, el 57,5% de los aprendices encontraron la mejor solución y hay otras 21 soluciones diferentes. En el nivel 12, los valores son 55,2% y 25, respectivamente. Las equivalencias en estos niveles son más sutiles y no indican necesariamente código incorrecto, sino combinaciones interesantes de alternativas y secuencias que resultan en caminos idénticos, o variantes de giros que combinan de distintas maneras. Por ejemplo, en el nivel 11 el código estándar `[rep [ad si-i [gi]]` fue escrito por 1.015 aprendices; 204 proponen `[rep [ad ad si-i [gi]]`, que toma ventaja de que la repetición pueda hacerse en pares debido a las características particulares del laberinto (figura 3.17). Otros 135 aprendices propusieron `[rep [ad si-i [gi ad]]`.

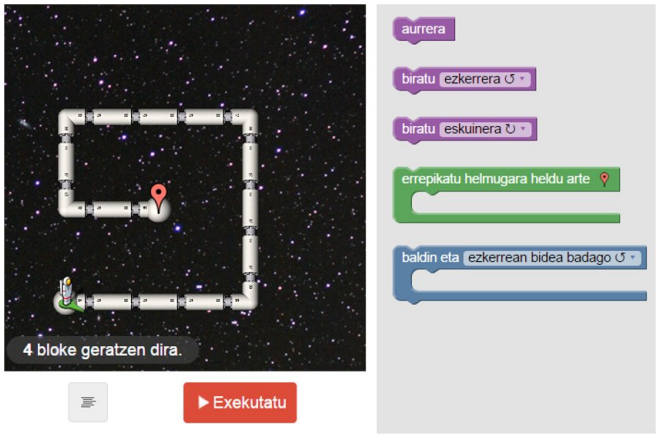


Figura 3.17. Nivel 12 de Kodetu.

El nivel 13, donde la longitud de código se mueve entre 6 y 10 bloques, es el que más expresividad permite en Kodetu. La solución más común, `[rep [si-ad [ad] si-i [gi] si-no gd]`, fue utilizada solo por el 13,4% de los aprendices. Después aparece una larga lista de otras 179 soluciones. Esta gran variedad de posibilidades, en un contexto donde un código diferente no es necesariamente peor, nos permite clasificar a los aprendices de acuerdo a la originalidad relativa de su código, considerando el número de aprendices que encontraron la misma solución como un percentil. Asignamos este valor a cada código y desarrollamos gráficos de dispersión que relacionan esta originalidad con otras características. En la figura

3.18 representamos este valor en el eje X (a la izquierda los participantes que hacen códigos más comunes, hacia la derecha los que hacen códigos menos comunes) y lo comparamos con las medias de las tres variables fundamentales: número de ejecuciones, número de cambios y tiempo invertido. Descubrimos que los participantes que tienen códigos más originales tienden a comportarse mejor (el 77,5% de los participantes hacen menos ejecuciones, el 67,5% hacen menos cambios y el 66,7% necesitan menos tiempo). Se puede medir la expresividad de cada persona en su capacidad de desarrollo de código, aún en un entorno limitado y en un contexto de aprendizaje.

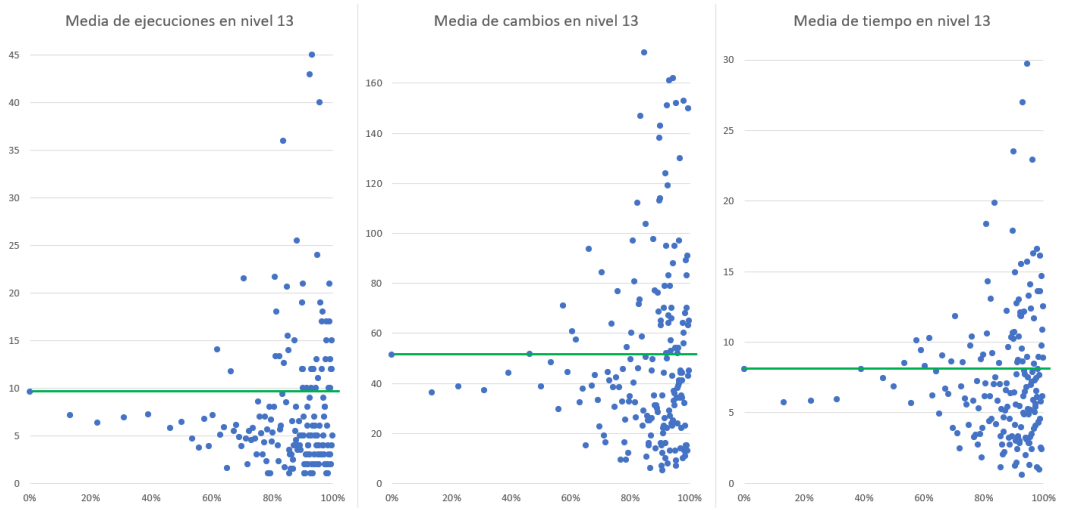


Figura 3.18. Relación entre originalidad relativa de código (eje x) y en el eje y: número de ejecuciones (izquierda), número de cambios (centro) y tiempo de nivel (derecha, en mins.) en el nivel 13 de Kodetu. La línea verde marca la media para el grupo más numeroso de participantes con el mismo código.

El nivel 14 es similar, pero con un rango menor de variedad porque el código está limitado entre 4 y 7 bloques. Solo el 22,5% de los aprendices eligen la solución más común, que en este nivel se aprovecha de darse cuenta de que moverse hacia adelante y girar a la izquierda solo cuando no se puede hacia adelante soluciona el reto en cuatro bloques: `[rep [si-ad [ad] si-no [gi]]]`.

Al diseñar los niveles 13 y 14 consideramos que la solución más conveniente del nivel 13 soluciona también el nivel 14. Nos preguntamos en qué medida los aprendices se habrán percatado de eso. Observamos los algoritmos propuestos que solucionan ambos niveles, encontrando que el 32,9% de los aprendices que solucionan el nivel 14 pudiendo hacerlo con el mismo código del 13 lo hacen (a pesar de que hay maneras más rápidas de hacerlo). Es importante notar que el código no se guarda en Kodetu: debe reescribirse desde cero en cada nivel. En el sentido inverso, el 18,2% de los aprendices cuya solución del nivel 13 no resuelve el 14 intentan repetir el código sin éxito.

Repetir el código entre niveles no es tan común en el nivel 15. Solo el 6% de los aprendices escribieron código tan general en el nivel 14 que también podría ser

utilizado para resolver el 15. Alrededor del 10% de esos aprendices se dieron cuenta de esto y repitieron su código.

Por otro lado, la dificultad de este nivel se concreta en una gran variedad de soluciones. La solución más común solo fue utilizada por el 14% de los aprendices.

Evaluamos también la originalidad relativa del código observando la evolución de cada aprendiz entre los distintos niveles, preguntándonos si los aprendices con código más original son siempre los mismos o no. Pocos aprendices comparten códigos originales en ambas series, lo que muestra que la aproximación que debe utilizarse en programación secuencial pura es significativamente diferente a la que permiten los algoritmos con estructuras de control.

3.3.3. Progreso de la dificultad entre niveles

El tercer aspecto de esta fase de la investigación busca identificar las variables que puedan ayudar a definir estrategias que adapten este tipo de plataformas según el aprendiz progresa. Aquí, suministrar una progresión mantenida y suave de dificultad es un aspecto fundamental (Gujberova y Kalas, 2013). En Kodetu ajustamos la dificultad equilibrando complejidad de laberinto, disponibilidad de bloques, límite de bloques y límite de tiempo (de forma externa a la plataforma).

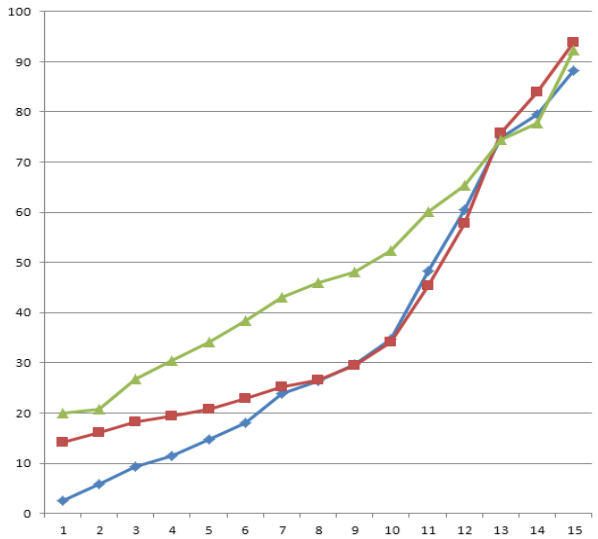


Figura 3.19. Evolución del ratio de fallo en los grupos A (rombos azules), B (cuadrados rojos) y C (triángulos verdes).

La figura 3.19 muestra que la evolución del ratio de fallo es estable en Kodetu cuando no hay límites de tiempo (grupo C). En los grupos con esos límites (A y B), la dificultad se incrementa en una medida diferente en tres secuencias: 1) niveles 1 a 10; 2) niveles 10 a 13; 3) niveles 13 a 15. Esos puntos de inflexión deberían considerarse cuando se usan plataformas para mejorar el porcentaje de éxito. Adicionalmente, los desarrolladores de este tipo de plataformas podrían usar

variables de niveles de grupos, como el número de intentos medio, o el tiempo medio utilizado para resolver un nivel, para detectar cuándo un aprendiz puede necesitar ayuda para resolver un reto. En el aspecto individual, el número de cambios realizados al espacio de trabajo o el número de bloques retirados podrían identificar aprendices atascados en un nivel, de cara a suministrar alguna pista, o a adaptar el siguiente reto.

3.3.4. Otros resultados

Analizando las interacciones de los aprendices en cualquiera de los momentos de la evolución del espacio de trabajo, encontramos un total de 31.921 códigos que resolvían el nivel correspondiente. De esos, 2.458 (el 7,7%) no fueron reconocidos por los aprendices como soluciones correctas, de modo que realizaron cambios adicionales para finalmente resolver el nivel. 110 aprendices de estos (el 0,3%) finalmente no resolvieron el nivel.

El análisis del código probado por los aprendices en el nivel 15 muestra soluciones inesperadas y pintorescas. La cuarta solución más utilizada fue `[rep [si-ad [ad] si-no [gd] si-i [gi]]]` y la octava solución más utilizada `[rep [si-ad [ad] si-no [gi gi] si-d [gd ad] si-no [si-i [gi]]]]`. Se encuentran también algunas soluciones excelentes que usan solo cinco bloques, como `[rep [si-i [gi ad] si-no [gd]]]` y `[rep [si-ad [ad gd] si-no [gi]]]`.

El análisis de las largas colas muestra información interesante sobre la eficiencia del código. En el nivel 7 la longitud media del código de las soluciones no principales tiene 4,16 bloques adicionales, lo que se convierte en 3,63 pasos más para el astronauta en llegar a la meta (lo que sería equivalente al tiempo de ejecución). Contrariamente a los códigos secuenciales, en el nivel 15 1,77 bloques de media adicionales se convierten en 0,70 pasos menos (peor longitud de código, pero mejor tiempo de ejecución).

También hemos analizado la longitud del camino que sigue el astronauta (el número de instrucciones de movimiento que "ejecuta"). En el nivel 15, algunas soluciones no buscan el código más sencillo sino el camino más corto. Hay hasta 20 soluciones (el 5,7%) que acaban el laberinto en solo 11 pasos, a pesar de que no es la opción recomendada por las instrucciones dadas en pantalla.

3.4. Conclusiones y discusión

Los resultados presentados en este capítulo ayudan a entender los primeros pasos de programadores aprendices y pueden servir también de consideración para diseñar mejores herramientas de PC. Revisamos las conclusiones asociadas a nuestros objetivos de investigación.

PI_K1. ¿Cuál es la influencia de las características personales (sexo, edad, afinidad tecnológica y conocimiento de programación previo) en las primeras experiencias del PC?

Encontramos que la edad influye significativamente en el rendimiento en estos desafíos, mejorando según avanza hasta 13-14 años, aunque a partir de ese punto el rendimiento empeora. El sexo influye también en el rendimiento: los chicos resuelven los retos mejor que en las chicas en los primeros niveles, y al hacerse más abstractos con la incorporación de las estructuras combinadas, esta tendencia se invierte. El conocimiento previo de programación no parece ser una ventaja ya que en algunos indicadores su comportamiento es peor; tampoco la afinidad tecnológica es un buen predictor del desempeño. Considerando que la edad influye significativamente en el rendimiento en estos desafíos, es recomendable adaptar el conjunto de retos si los aprendices tienen edades de educación primaria o primer ciclo de secundaria (por debajo de 14 años). Al respecto del sexo, las chicas necesitarían más ayuda o refuerzo en los primeros niveles, los chicos en los retos con estructuras de control complejas.

PI_K2. ¿Cuántas soluciones diferentes encuentran los aprendices en sus primeros retos de programación y qué se puede concluir de estas diferencias?

A pesar de las fuertes limitaciones que propone Kodetu en los niveles con bloques estructurados, existe una importante diversidad de soluciones que permite identificar la relativa originalidad de los aprendices, así como su habilidad para reutilizar soluciones cuando es posible. La originalidad de código se manifiesta en diferentes aprendices en los niveles secuenciales que en los estructurados. La plataforma podría detectar desviaciones desde los valores más frecuentes, tanto en casos positivos (código diferente pero eficiente) como en los negativos (diversidad causada por ineficiencia de código) y ofrecer sugerencias adecuadas.

PI_K3. ¿Qué variables podrían ayudar a plantear estrategias de personalización y adaptación de este tipo de herramientas de aprendizaje del PC?

Tanto la edad como el sexo permiten plantear personalizaciones de las herramientas, así como la originalidad relativa del código según los aprendices avanzan en los retos. Hemos encontrado dos grandes obstáculos que limitan el progreso de nuestros aprendices: el límite de tiempo y las estructuras de control implicadas en cada reto. Considerando esto, podríamos ajustar dinámicamente el tiempo disponible o el conjunto de retos presentado a cada aprendiz, de acuerdo a sus características iniciales y a la manera en la que va resolviendo los retos anteriores.

Es importante conocer y comentar las limitaciones de esta parte de nuestro estudio. A pesar de que se recomienda que el trabajo sea individual, en los grupos supervisados no se puede evitar que algunos aprendices interactúen entre ellos y

esta cooperación muy probablemente influye en los resultados, no sabemos en qué medida. Además, a pesar de ser un número muy sustancial de aprendices, todos ellos son de una única zona geográfica (Bizkaia y, en menor medida, Gipuzkoa y Araba).

En cuanto a los aprendices no supervisados, hemos revisado los registros para eliminar datos incorrectos, pero la propia naturaleza de Kodetu y su aproximación anónima hacen posible que algunos aprendices hayan realizado el juego total o parcialmente dos veces en diferentes sesiones y sean tratados estadísticamente como si fueran dos personas diferentes.

En cualquier caso, consideramos que el volumen de datos y el preproceso de filtrado minimizan el impacto de estas limitaciones en los resultados finales de nuestro análisis.

Estimamos que el volumen de información obtenida es muy significativo y que la definición cuidadosa de recogida automática de información de interacción es un buen punto de partida para esta y futuras investigaciones. Para facilitar la colaboración con la comunidad científica, publicamos el conjunto de datos completo descrito en este capítulo con acceso abierto a través del *Open Science Framework*: https://osf.io/qjrs9/. Hasta nuestro conocimiento, no existía en la fecha otro estudio que hubiera liberado algún registro detallado con más de un millón de interacciones de forma abierta.

Tras finalizar esta parte de nuestra investigación, Kodetu seguía encontrándose *online* y abierta, disponible en inglés, español y euskera. Sin embargo, nos planteamos la relevancia de variar la plataforma para obtener resultados más concluyentes en alguna de las líneas de interés: la posibilidad de personalizar la plataforma de acuerdo al comportamiento de los aprendices, la incorporación de ludificación y sus posibilidades para ampliar la motivación y el rendimiento, la eliminación de los límites de bloques en todos o algunos de los niveles para aumentar la expresividad del código planteado por los aprendices, etc.

Entre 2016 y 2017 desarrollamos varias versiones prototipo que exploraban algunas de estas opciones (en particular, Kodetu con ludificación, Kodetu sin límites y Kodetu con retos personalizables). Pero la dificultad de abordar todas estas posibilidades en nuestra investigación nos forzó a elegir una de las opciones como la más relevante: la generalización.

If something that you are doing does not challenge you, then it does not change you.

Anónimo

La felicidad consiste en tomar con alegría lo que la vida nos da y en soltar con la misma alegría lo que la vida nos quita.

San Agustín

Capítulo

4

Generalización en Kodetu

Tras la primera fase de nuestra investigación utilizando Kodetu, nos dimos cuenta de que la abstracción que se necesita en los niveles superiores del juego es un aspecto clave. Sin embargo, tal y como habíamos observado, la mayoría de las plataformas formativas de PC no abordan este tema de una forma explícita.

De esta forma, nos planteamos la posibilidad de incluir la generalización para observar su impacto en el aprendizaje del PC en los retos de Kodetu. Con esta idea en mente, comenzamos en 2016 a diseñar una nueva versión de la herramienta, KodetuGen (KG), que da inicio a la segunda fase de nuestra investigación, comentada en este y los próximos capítulos.

4.1. Objetivos de investigación

Completando los objetivos de investigación comentados en el capítulo anterior, seguimos con la intención de entender los procesos de aprendizaje del PC en aprendices de programación con un juego *online* basado en retos, con la incorporación específica de la generalización como una competencia relevante del PC. Utilizando el nuevo sistema KodetuGen, nos planteamos las siguientes preguntas de investigación para esta fase:

PI_G1 ¿Permite el análisis de las interacciones de un aprendiz en un sistema de estas características entender el desarrollo del PC?

PI_G2 ¿Puede incluirse la competencia de generalización en los retos de programación de las plataformas lúdicas de aprendizaje autónomo de una manera eficaz?

PI_G3 ¿Es la generalización un aspecto de influencia significativa en el desarrollo del PC? ¿Cómo impacta en el aprendizaje?

PI_G4 ¿Cómo influyen las características personales (sexo, edad, afinidad tecnológica, y conocimiento de programación previo) en las primeras experiencias del PC y en su relación con la generalización?

PI_G5 ¿Afecta la incorporación de la generalización a las soluciones encontradas por los aprendices?

PI_G6 ¿Qué consideraciones deben tenerse en cuenta para diseñar este tipo de sistemas?

4.2. Metodología

4.2.1. Diseño de KodetuGen

Con el objetivo fundamental de incorporar la generalización como un elemento explícito en la actividad de Kodetu, consideramos el fundamento de esta habilidad: "abstraer lo que es común y esencial a muchas cosas, para formar un concepto general que las comprenda todas" (RAE). En nuestro juego, donde el aprendiz debe diseñar un código para resolver un laberinto, planteamos que la generalización se concrete en diseñar un solo código que pueda resolver **varios** laberintos. De esta forma, siguiendo el enfoque de complejidad progresiva de los niveles, creamos el concepto de **laberinto dual**: dos laberintos diferentes que puedan ser solucionados con el mismo código.

Esto puede realizarse mediante mecanismos tan sencillos como rotar el laberinto (parece distinto, pero es el mismo laberinto), pero también a partir de consideraciones más abstractas como encontrar el algoritmo que resuelva un laberinto cualquiera. Intentamos escalar la dificultad de la misma manera en la que crece la dificultad algorítmica en los niveles de Kodetu.

Con este planteamiento de base, consideramos los siguientes elementos principales para replantear la plataforma:

- Para permitir el progreso a un aprendiz que no conoce la mecánica básica, incluimos los laberintos duales en los niveles 3 al 15, tras el primer nivel tutorial y el segundo de introducción.
- Para facilitar el análisis comparativo de los aprendices que jueguen en las dos plataformas, mantenemos el laberinto original de Kodetu en cada nivel como primero, y añadimos un segundo laberinto que incluye la generalización en función de los bloques disponibles en cada caso.
- Desechamos la opción de navegación libre y mantenemos solo la secuencial (hace falta acabar correctamente un nivel para pasar al siguiente).

La incorporación del segundo laberinto nos obliga también a replantear la interfaz operativa en función del nuevo concepto de laberinto dual:

- Se incorpora un botón de ejecución independiente en cada laberinto, para que el aprendiz pueda ejecutar el código por separado en cada uno de ellos.
- El botón de ejecución principal se mantiene, pero ahora lanzará el código primero con el laberinto superior y después, si es exitoso, con el inferior.
- Tras constatar que el proceso de observación del camino de ejecución se alarga bastante al tener que recorrer dos laberintos, añadimos un segundo botón de ejecución *rápida* que realiza la ejecución de los bloques de código en los dos laberintos al doble de velocidad.
- Se elimina la parte de la encuesta inicial relacionada con las sensaciones sobre la tecnología y la profesión tecnológica, que hacían más tedioso el proceso de registro y no estaban siendo significativas en la investigación.

Con estas consideraciones de diseño, describimos a continuación los retos que plantean los 15 niveles de KodetuGen. Como ya hemos comentado, los niveles 1 y 2 permanecen como en Kodetu, con las soluciones esperables `[ad ad]` y `[ad ad gd ad]`, respectivamente (mantenemos las abreviaturas introducidas en el capítulo 3).

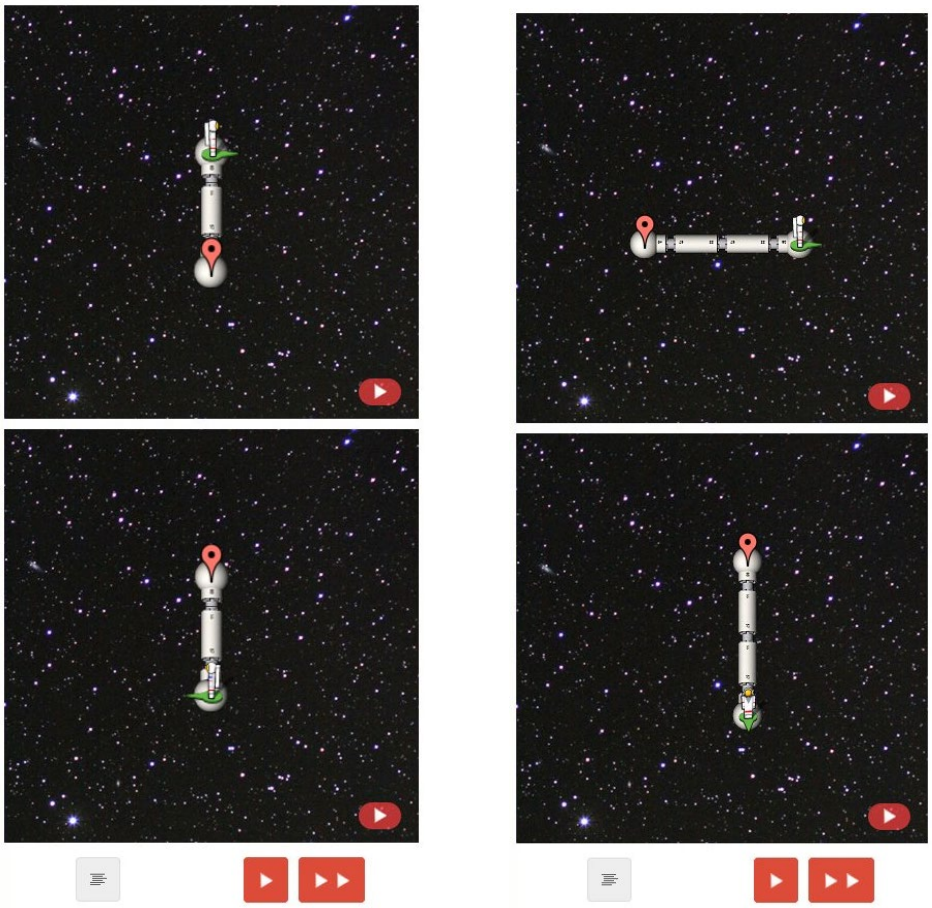


Figura 4.1. Laberintos de nivel 3 (izquierda) y 4 (derecha) de KodetuGen. Cada laberinto tiene su propio símbolo de ejecución y en la parte inferior se observan los botones de ejecución generales, con dos velocidades.

El nivel 3 añade ya el primer laberinto inferior, que es el mismo girado 180º, de modo que el camino a recorrer es idéntico (figura 4.1 izquierda). Se muestra una pista *"Ahora tienes que conseguir que el mismo programa ayude a los dos astronautas"*. Solución, al igual que en Kodetu: `[gd ad ad]`.

El nivel 4 (figura 4.1 derecha) añade un laberinto dual girado 90º con respecto al original, con solución: `[gi gi ad ad ad]`.

El nivel 5 se resuelve de nuevo con un giro de 180º: los caminos son visualmente invertidos, pero los giros a realizar son idénticos (figura 4.2 izquierda). Solución: `[ad gi ad gd ad]`.

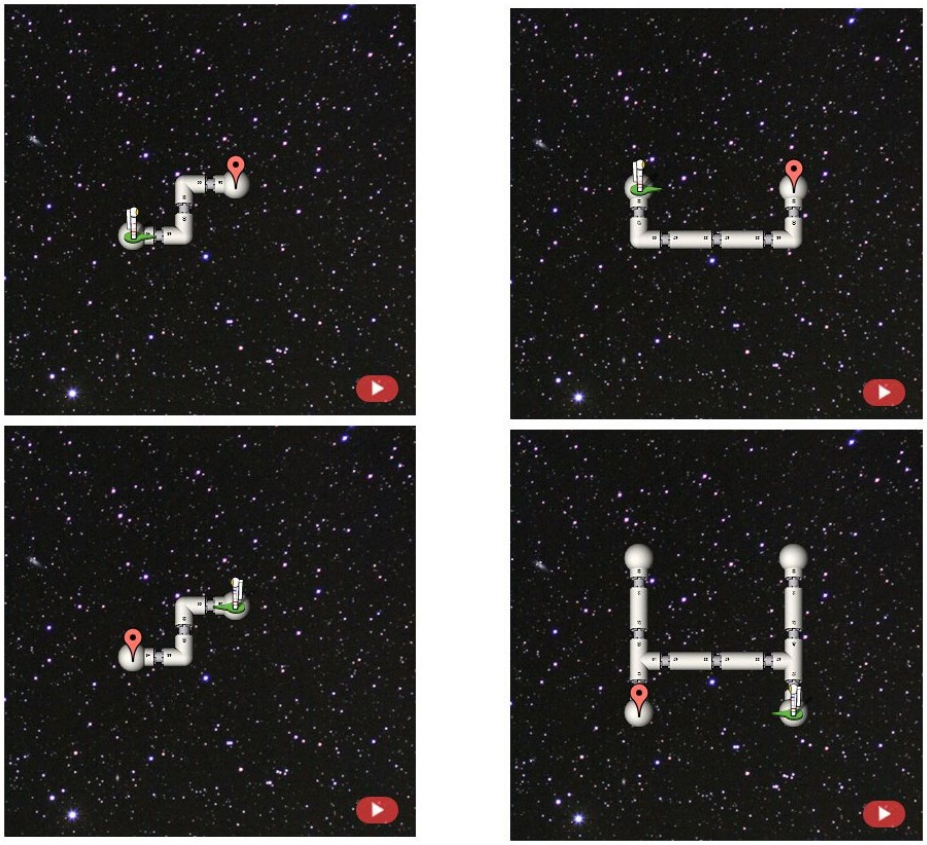


Figura 4.2. Laberintos de nivel 5 (izda.) y 6 (dcha.) de KodetuGen.

El nivel 6 (figura 4.2 derecha) propone por primera vez un laberinto inferior diferente en forma al superior. Solo con secuencias no pueden hacerse caminos distintos: realmente es una dificultad visual de diferencia de forma, pero el camino a recorrer es el mismo, girado 180º. Solución: `[gd ad gi ad ad ad gi ad]`.

El nivel 7 (figura 4.3 izquierda), último con solo secuencias, plantea el mismo camino sobre el mismo laberinto, pero cambiando los puntos de salida y llegada, de modo que el aprendiz deba reconocer cuál de todos los caminos posibles es el que resuelve ambos laberintos. Solución: `[ad ad ad ad gd ad ad ad ad gd ad ad gd ad ad]`.

El nivel 8, como el de Kodetu, es simplemente de introducción a las repetitivas y los límites de código en 2 bloques (figura 4.3 derecha). Lo resolvemos con un giro de 90º en el laberinto dual. Solución única: `[rep [ad]]`.

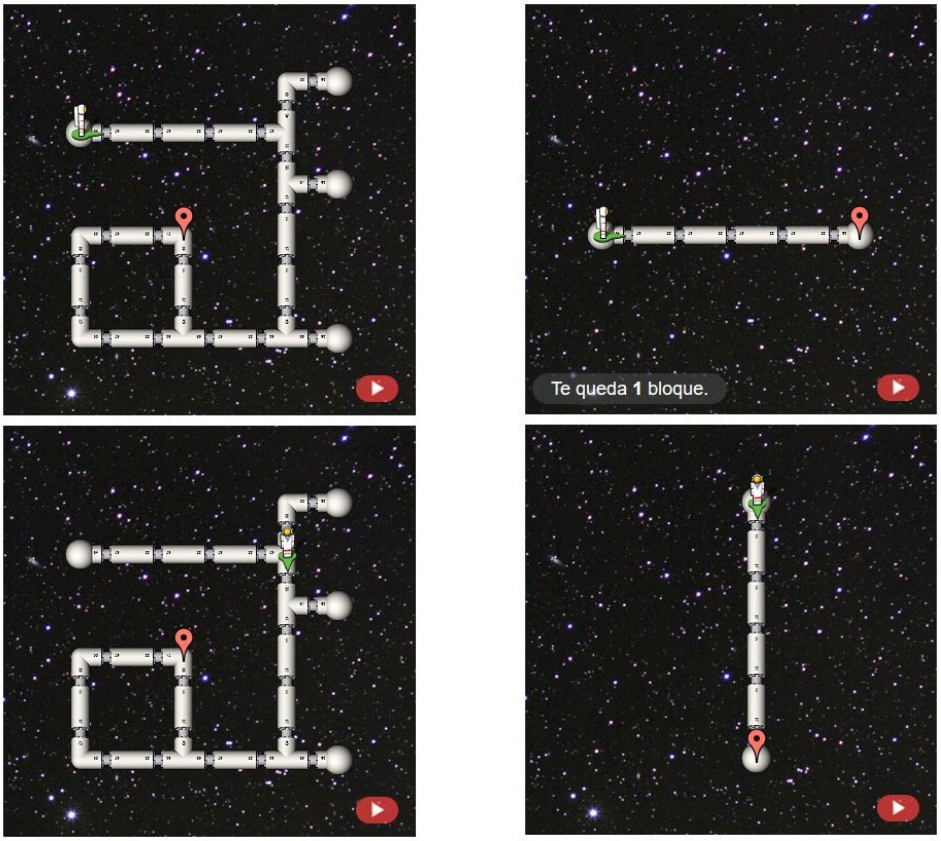


Figura 4.3. Laberintos de nivel 7 (izda.) y 8 (dcha.) de KodetuGen.

El nivel 9 (figura 4.4 izquierda) plantea la generalización de secuencias repetidas, con un zigzag de cuatro pasos que hay que repetir de forma especular. Ampliamos ligeramente el límite (6 bloques en vez de 5) para poder valorar aquí si el segundo laberinto induce a alguna modificación del algoritmo más probable. También las longitudes de los caminos a recorrer son diferentes, para introducir el concepto de que el número de repeticiones no depende del código, sino del espacio en el que se ejecuta. Solución más eficiente: `[rep [ad gi ad gd]]`.

El nivel 10 (figura 4.4 centro) mantiene el anterior nivel 10 de Kodetu, modificando el camino repetido tras la secuencia inicial con diferente longitud. Se refuerza este concepto con una pista *"Los dos astronautas tienen que llegar con las mismas instrucciones, aunque los caminos no sean iguales"*. Se mantiene la limitación de 5 bloques, que hace la solución única: `[ad ad gi rep [ad]]`.

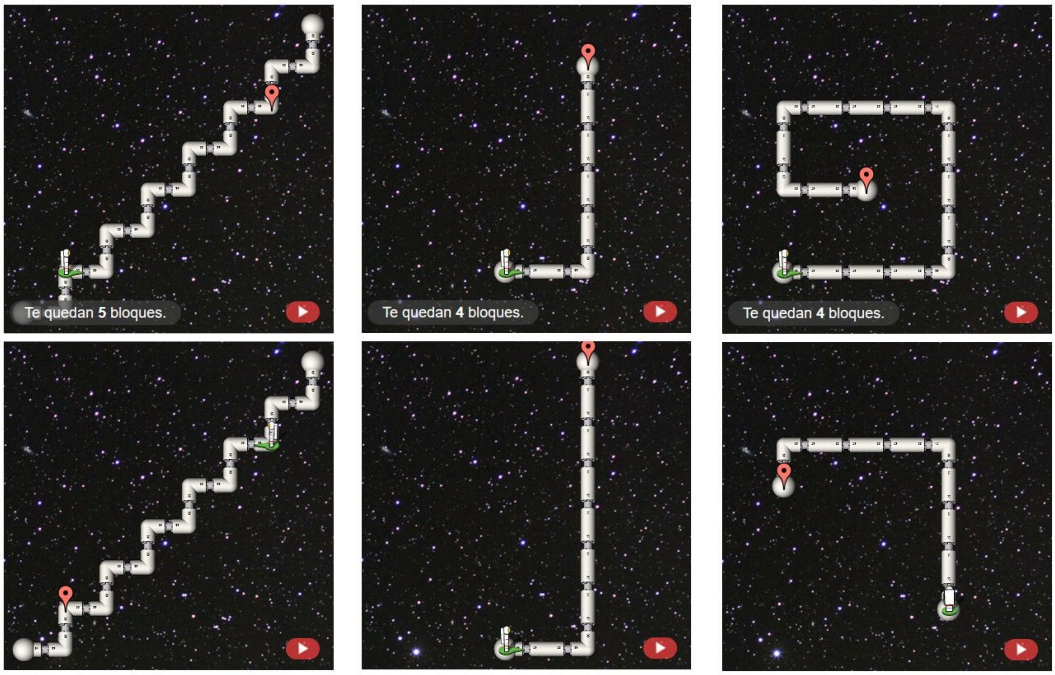


Figura 4.4. Laberintos de niveles 9 (izda.), 10 (centro) y 11 (dcha.) de KodetuGen.

El nivel 11 incorpora, como en Kodetu, la estructura alternativa de una rama (figura 4.4 derecha). En el laberinto dual no solo es diferente el número de pasos que hay que avanzar en cada parte del camino, sino que también lo es el número de giros. Solución más habitual: `[rep [ad si-i [gi]]]`.

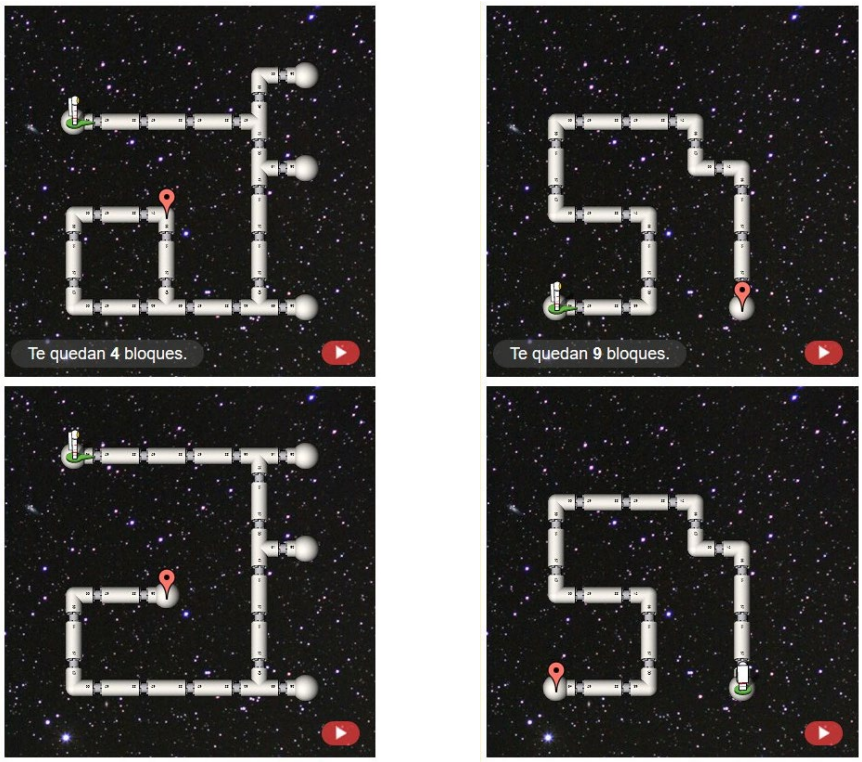


Figura 4.5. Laberintos de niveles 12 (izda.) y 13 (dcha.) de KodetuGen.

En el nivel 12 (figura 4.5 izquierda) se refuerza la generalización de repetitivas con alternativas, con un laberinto que permite varios caminos manteniendo el límite de 5 bloques. Solución posible: `[rep [ad si-d [gd]]]`.

El nivel 13 (figura 4.5 derecha) incorpora las alternativas dobles y obliga a mayor anidamiento de estructuras con giros a ambos lados. Optamos por resolver el mismo laberinto al revés, con giros invertidos. Mantiene la mayor expresividad de código de Kodetu, con límite de 10 bloques. Solución posible: `[rep [si-ad [ad] si-no [si-i [gi] si-no [gd]]]]`.

El nivel 14 es una generalización con un laberinto más complejo, donde el laberinto inferior evita la solución más trivial que en Kodetu consistía solo en ir hacia adelante y a la izquierda cuando no hay más opciones (figura 4.6 izquierda). Posible solución: `[rep [si-ad [ad] si-i [gi] si-no [gd]]]`.

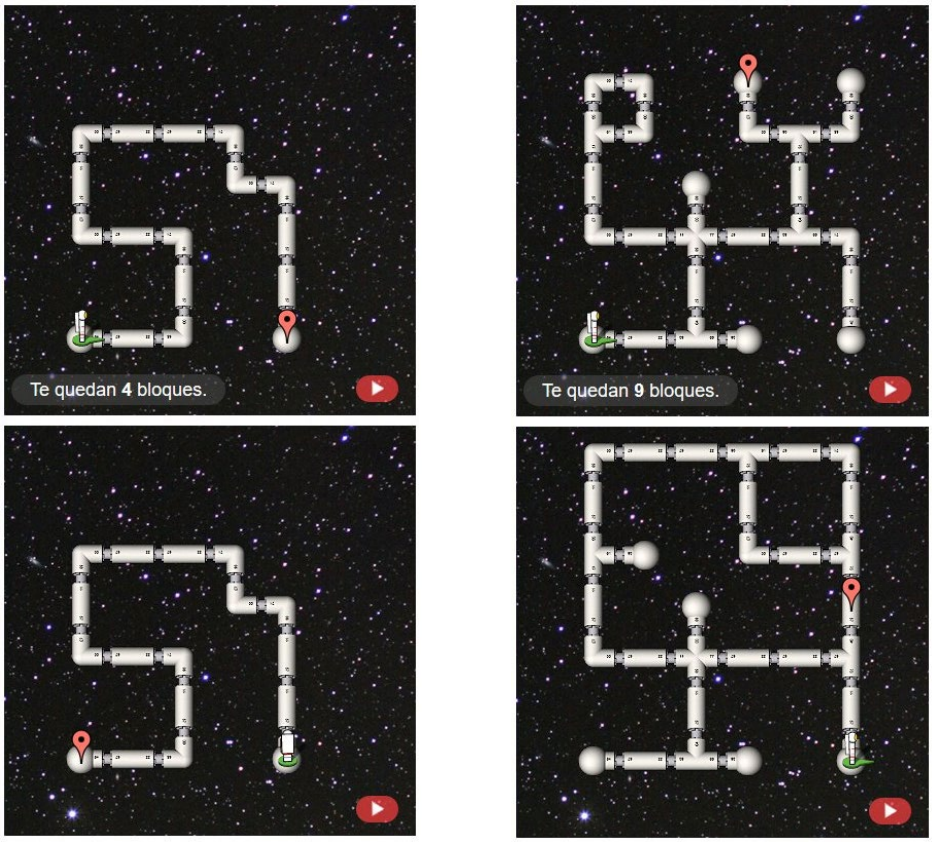


Figura 4.6. Laberintos de niveles 14 (izda.) y 15 (dcha.) de KodetuGen.

Por último, el nivel 15 (figura 4.6 derecha) mantiene su condición de laberinto más complicado, con un bucle que obliga a repensar el orden de comprobación de las tres opciones (adelante, izquierda, derecha). Para que quede claro que esa necesidad obliga a caminos que no son siempre los ideales, el inferior plantea un camino que podría solucionarse de forma trivial simplemente avanzando. Una posible solución es `[rep [si-ad [ad] si-no [gi] si-d [gd]]]`.

Diseñamos estos dos últimos niveles con una pequeña recompensa al final, pidiendo el nombre y personalizando un certificado digital que los aprendices pueden descargarse, "Amigo Kodetu" en el nivel 14 y "Experto Kodetu" en el 15 (figura 4.7).

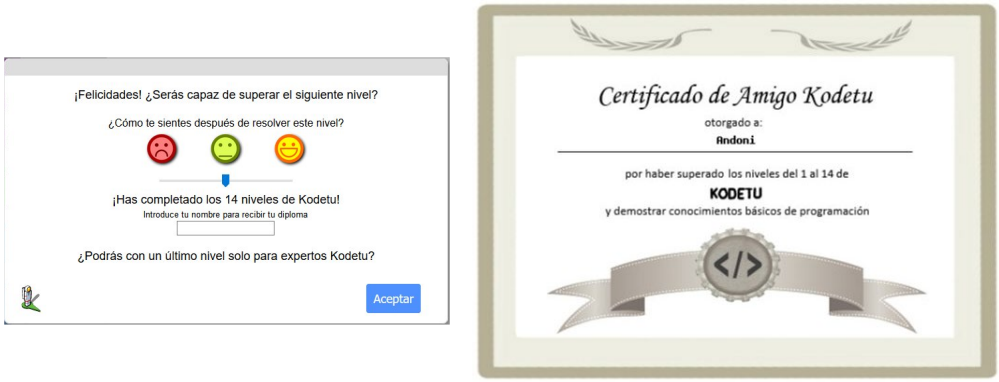


Figura 4.7. Ventana de petición de nombre (izda.) y diploma de éxito (dcha.) en nivel 14.

El resto de la interfaz, con los comentarios reseñados al respecto de los botones de ejecución, se mantiene como en Kodetu, con las mismas tres zonas (figura 4.8).

Figura 4.8. Aspecto general del interfaz de KodetuGen.

4.2.2. Modificaciones en KodetuWin

En nuestra herramienta de análisis añadimos la capacidad de gestionar una versión adicional de datos. Además de Kodetu lineal y con saltos, ahora puede recibir registros de KodetuGen. En este caso, además, rediseñamos el sistema interno de simulación de ejecución de código ya que, al haber laberintos duales a partir del nivel 3, tenemos que poder simular la ejecución del mismo código contra uno y otro laberinto, almacenando además por duplicado todos los indicadores de

eficiencia de código: longitud de camino, distancia a la meta, número de pasos de ejecución, etc.

4.2.3. Despliegue del experimento

Mantuvimos la plataforma KodetuGen activa desde septiembre de 2016 a diciembre de 2018. En ese lapso, se registraron accesos de 964 usuarios únicos, que realizaron 650.421 interacciones. Como con Kodetu, organizamos sesiones controladas con escolares, y además la plataforma de forma abierta recogió interacciones de usuarios libres, tanto relacionados con centros educativos como no.

4.3. Resultados y conclusiones

Ya en los primeros meses de experimentación con KodetuGen empezamos a notar en la propia observación en los grupos controlados un comportamiento que nos hizo plantearnos la conveniencia de las decisiones tomadas. Una mayoría de aprendices solo se preocupaban de resolver el laberinto superior, y con ese código se resolvía siempre el inferior.

Si bien la misma presencia del laberinto inferior aporta el concepto de generalización y debería tener influencia en el aprendizaje, este comportamiento, inesperado en un primer momento, nos llevó a una segunda iteración de rediseño. Aunque mantuvimos la experimentación y tenemos un significativo volumen de datos que incluiremos posteriormente en el análisis comparativo, **decidimos desarrollar una versión 2 de KodetuGen** que afrontara el potencial de la generalización sin el inconveniente comentado.

De esta forma, revisaremos posteriormente este conjunto de datos y lo incluiremos en el análisis, pero daremos más importancia al producido en la segunda versión, ya que entendemos que los resultados de esta fase están en parte contaminados por la manera en la que una mayoría de aprendices se enfrentan a los retos.

Para validar esta decisión, hicimos un control temprano de soluciones de niveles de KodetuGen con respecto a las obtenidas en Kodetu. En nuestro diseño de investigación, la generalización debería influir significativamente en los códigos propuestos por los aprendices. Y nos encontramos que excepto en los niveles 13 y 14, más de un 80% de los aprendices encontraban soluciones idénticas a las planteadas en Kodetu. Si bien podría ser interesante el análisis particular de esos dos niveles, consideramos parcialmente fallida esta versión, al desarrollar muy parcialmente el potencial de la generalización en esta prueba del PC.

Partiendo de esta experiencia, diseñamos la siguiente versión de nuestra plataforma con los mismos objetivos. Tratamos de ella y sus resultados en el siguiente capítulo.

Sé que me quedan muchos nuevos comienzos en muchos días nuevos. Cada uno de esos días, una parte de mí morirá para que otras den a luz.

Clarisa Pinkola Estés, en "Mujeres que corren con lobos".

Capítulo

5

Mejoras en la generalización

Tras los resultados de nuestro primer experimento de generalización, vemos imprescindible mejorar la manera en la que se plantea el aspecto del doble laberinto para que los aprendices puedan afrontar esa característica con mejor resultado.

Para ello, diseñamos y desarrollamos en 2017 una nueva herramienta KodetuGen2 (KG2) con el propósito de solucionar algunos de los problemas encontrados, con variantes más significativas en los quince niveles, de cara a conseguir un análisis más pormenorizado de sus posibilidades con respecto al aprendizaje del PC y, en particular, de esa generalización introducida en KodetuGen.

5.1. Objetivos de investigación

Completando los objetivos de investigación comentados en capítulos anteriores, seguimos con la intención de entender los procesos de aprendizaje del PC en aprendices de programación con un juego *online* basado en retos, con la incorporación específica de la generalización como una competencia relevante del PC. Diseñamos, desarrollamos y probamos el sistema KodetuGen2 y lo pusimos en línea en 2018 para el público general, realizando una serie de experimentos controlados con escolares. Nuestras preguntas de investigación para esta fase son las ya indicadas en el capítulo anterior:

PI_G1 ¿Permite el análisis de las interacciones de un aprendiz en un sistema de estas características entender el desarrollo del PC?

PI_G2 ¿Puede incluirse la competencia de generalización en los retos de programación de las plataformas lúdicas de aprendizaje autónomo de una manera eficaz?

PI_G3 ¿Es la generalización un aspecto de influencia significativa en el desarrollo del PC? ¿Cómo impacta en el aprendizaje?

PI_G4 ¿Cómo influyen las características personales (sexo, edad, afinidad tecnológica, y conocimiento de programación previo) en las primeras experiencias del PC y en su relación con la generalización?

PI_G5 ¿Afecta la incorporación de la generalización a las soluciones encontradas por los aprendices?

PI_G6 ¿Qué consideraciones deben tenerse en cuenta para diseñar este tipo de sistemas?

5.2. Metodología

5.2.1. Diseño de KodetuGen2

Los elementos principales que consideramos para replantear la plataforma son los siguientes:

- Revisar los laberintos duales para que no siempre la solución plausible del primer laberinto resuelva el segundo, de modo que promueva en el aprendiz la reflexión en los dos en conjunto.
- Reestructurar la secuencia de laberintos para disminuir la parte de programación secuencial y dar más importancia a la programación estructurada, revisando el desarrollo lineal de la dificultad en esa parte.
- Permitir la identificación de cada usuario para limitar el problema de usuarios reentrantes con diferentes códigos y facilitar el proceso de acceso al sistema.
- Incorporar la gestión de grupos para identificar a los grupos experimentales y facilitar su gestión.

También se incorporan algunas decisiones relacionadas con la interfaz y el proceso de registro de interacciones, intentando que en lo posible la herramienta no sufra cambios sustanciales que invaliden o dificulten las comparaciones posteriores:

- Se simplifica la ejecución con solo un botón (como en la primera versión de Kodetu), que primero ejecuta un laberinto (superior) y, solo si es exitoso, ejecuta después el otro (inferior).
- Se registra la información de grupo y de versión de Kodetu, añadidas en esta versión.

El despliegue de niveles que se plantea mantiene los dos primeros niveles de presentación y entrenamiento en un solo laberinto (figura 5.1). Soluciones esperables: `[ad ad]` y `[ad ad gd ad]`, respectivamente.

Al igual que en capítulos anteriores, utilizamos las abreviaturas `ad` para adelante, `gd` para giro derecha, `gi` para giro izquierda, `si-ad si-d si-i si-no` para las estructuras condicionales y `rep` para las repetitivas.

Figura 5.1. Laberintos de niveles 1 (izda.) y 2 (dcha.) de KodetuGen2.

Como en KodetuGen, se introduce la generalización en el nivel 3 (con la explicación *"Ahora tienes que conseguir que el mismo programa ayude a los dos astronautas"*) (figura 5.2 izquierda). Solución: `[gd ad ad]`.

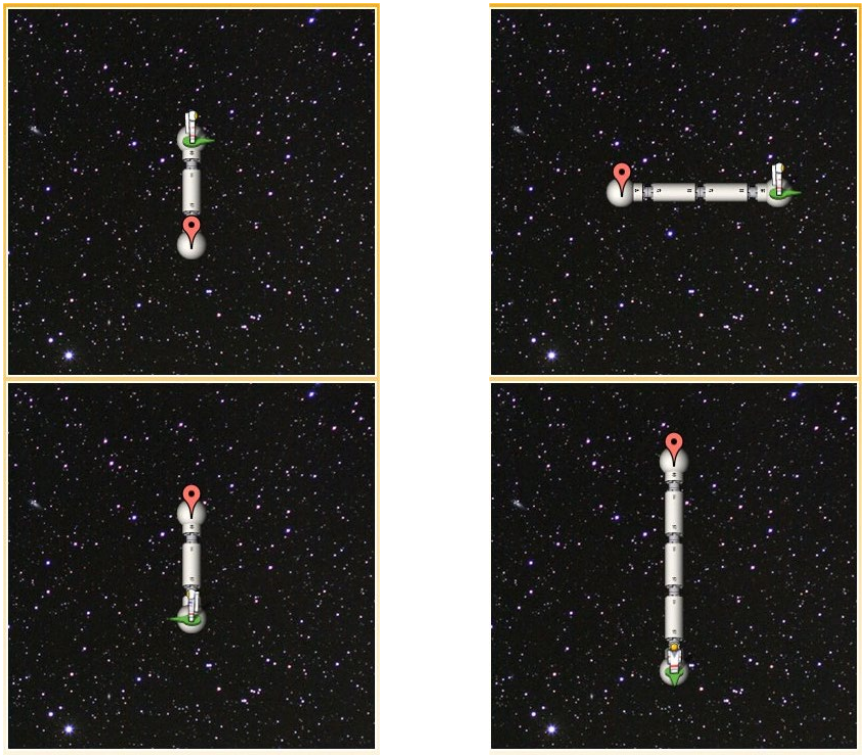


Figura 5.2. Laberintos de nivel 3 (izda.) y 4 (dcha.) de KodetuGen2.

La primera diferencia se establece en el nivel 4 (figura 5.2 derecha), donde además de plantear la secuencia de dos giros, la solución del primero no resuelve al segundo y hay que añadir un bloque para resolver el segundo laberinto. No es fácil plantear la generalización cuando aún no hay estructuras repetitivas y alternativas, pero aprovechamos la característica de que una vez que se llega al final del

laberinto puede haber instrucciones adicionales sobrantes. Estas solo se ejecutarán en el laberinto que corresponda, el más largo en este caso. Solución: `[gi gi ad ad ad ad]`.

El nivel 5 se mantiene como en KodetuGen y plantea una generalización con dos disposiciones especulares con caminos visualmente invertidos, pero giros idénticos (figura 5.3 izquierda). Solución: `[ad gi ad gd ad]`.

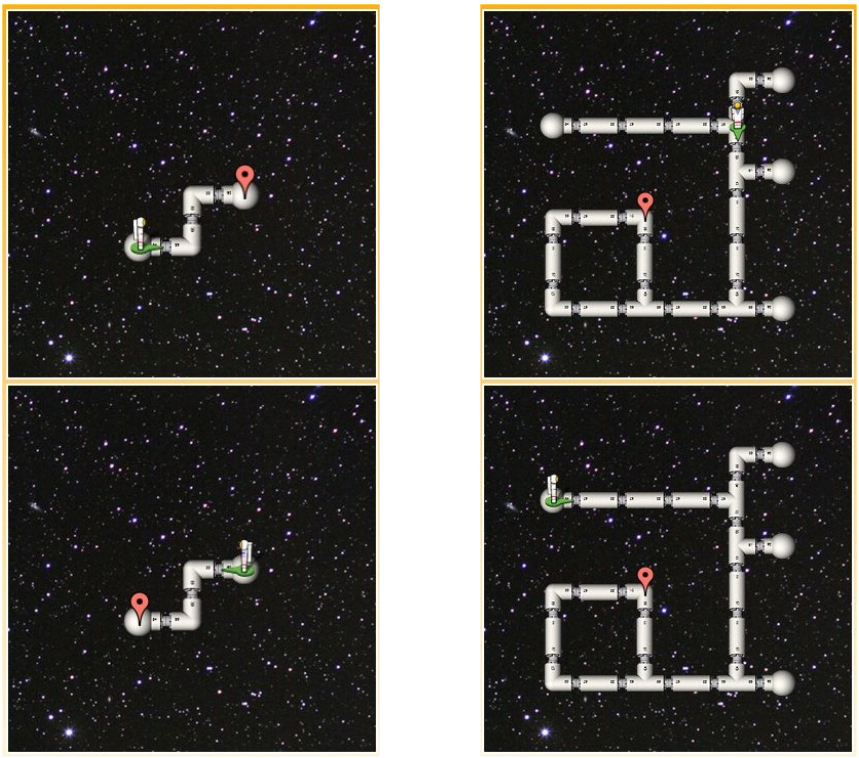


Figura 5.3. Laberintos de nivel 5 (izda.) y 6 (dcha.) de Kodetu Gen2.

La diferencia más importante se establece en el nivel 6, que se elimina en KodetuGen2, al valorarse que los niveles 6 y 7 cumplían objetivos similares. Esto nos deja un nivel más para desarrollar las estructuras, que -como hemos comentado- aportan los elementos de aprendizaje e investigación más relevantes. De esta forma, el nivel 6 de KodetuGen2 (figura 5.3 derecha) integra los niveles 6 y 7 de Kodetu, aprovechando un mismo laberinto con distintos puntos de salida y llegada para que el aprendiz deba reconocer cuál de todos los caminos posibles es el que resuelve ambos laberintos. Lo diseñamos especialmente para que el camino más corto del primer laberinto no resuelva al segundo. También hacemos que no sea siempre el primero en el que se empiece mirando a la derecha. Solución: `[ad ad ad ad gd ad ad ad ad gd ad ad gd ad ad]`.

El nivel 7 es simplemente de introducción a las repetitivas como el 8 de Kodetu y KodetuGen. Se inician los límites (2 bloques) y en esta versión incluimos los laberintos no solo con distinta orientación, sino con distinta longitud, para introducir el concepto de que una misma repetición de código puede cubrir distintas longitudes reales de camino (figura 5.4 izquierda). Solución: `[rep [ad]]`.

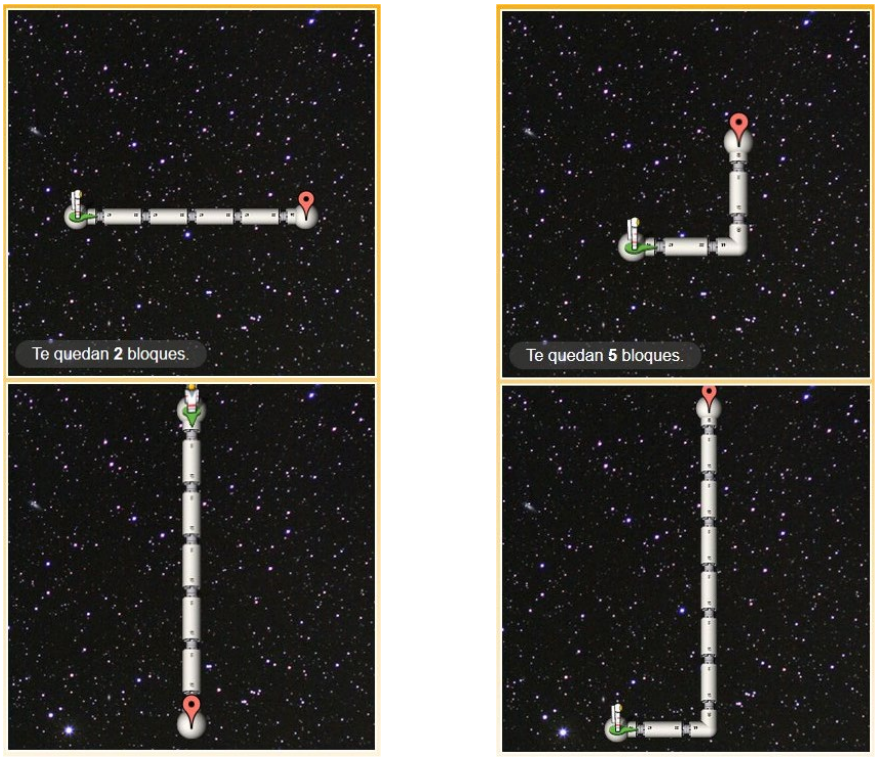


Figura 5.4. Laberintos de nivel 7 (izda.) y 8 (dcha.) de KodetuGen2.

El nivel 8 (figura 5.4 derecha) es equivalente al anterior nivel 10 de Kodetu, que se intercambia con el 9, al ser más natural la progresión por su menor complejidad. Se necesita una pequeña secuencia antes de la repetitiva y el camino repetido es de diferente longitud. También se mantiene la limitación de 5 bloques que hace que la solución sea única: `[ad ad gi rep [ad]]`.

El nivel 9, similar al anterior nivel 9, plantea la generalización de secuencias repetidas con un zigzag de cuatro pasos que hay que repetir de forma especular. Cambiamos el límite (8 bloques) para poder examinar aquí otras posibles soluciones que diseñan los aprendices (al plantear la repetitiva desde el inicio, o no). También las longitudes de los dos laberintos son diferentes, para reforzar que el número de repeticiones no influye en el código (figura 5.5 izquierda). Solución más eficiente: `[rep [ad gi ad gd]]`.

El nivel 10 (figura 5.5 central) es una propuesta nueva que plantea el mismo laberinto, con la única diferencia de un giro inicial que debe hacerse antes de la repetición, pero además sin límites, para observar en qué medida hay aprendices que intentan volver a la solución secuencial en cuanto se les permite, o incorporan ya el uso de las estructuras repetitivas como una mejor manera de plantear este tipo de algoritmos. Solución más eficiente: `[gi rep [ad gi ad gd]]`.

El nivel 11, al igual que en las dos plataformas anteriores, incorpora la estructura alternativa de una rama. En el planteamiento ahora sin embargo no solo es diferente el número de pasos que hay que avanzar en cada parte del camino, sino que también lo es el número de giros, para que en el aprendizaje se observe como

toda la estructura se debe repetir un número diferente de veces, tanto los giros como los avances (esto obliga a hacer el giro dentro de la estructura repetitiva y no previamente). Se amplía el límite a 8 bloques para permitir mayor expresividad en el código (figura 5.5 derecha). Solución más habitual: `[rep [ad si-i [gi]]]`.

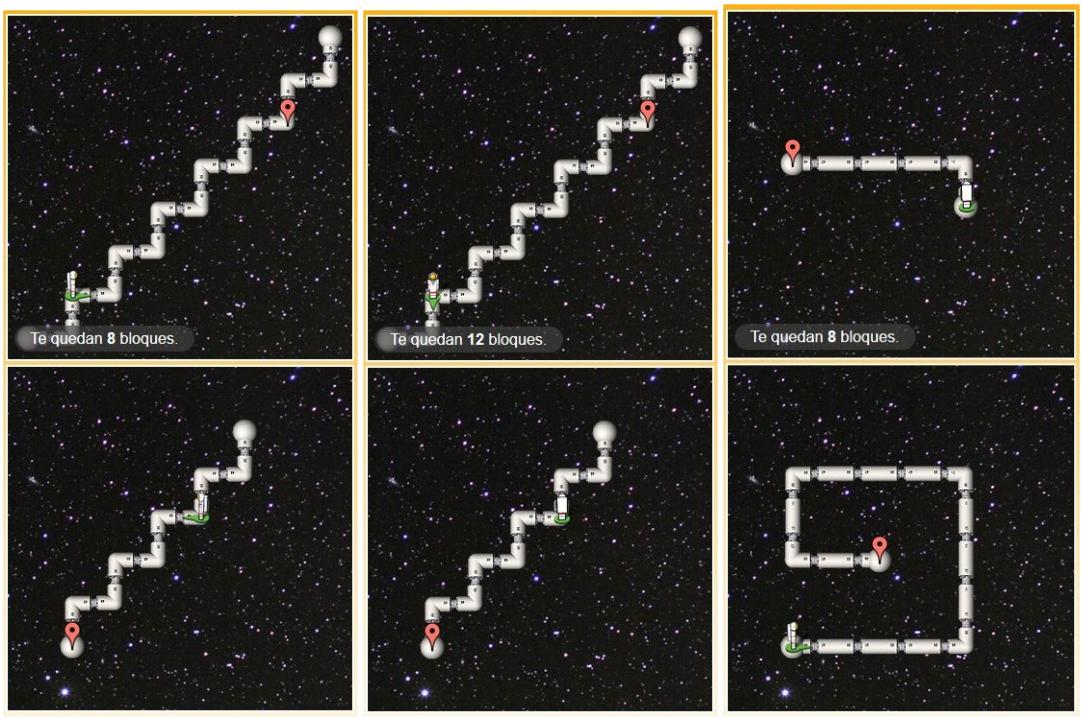


Figura 5.5. Laberintos de niveles 9 (izda.), 10 (centro) y 11 (dcha.) de KodetuGen2.

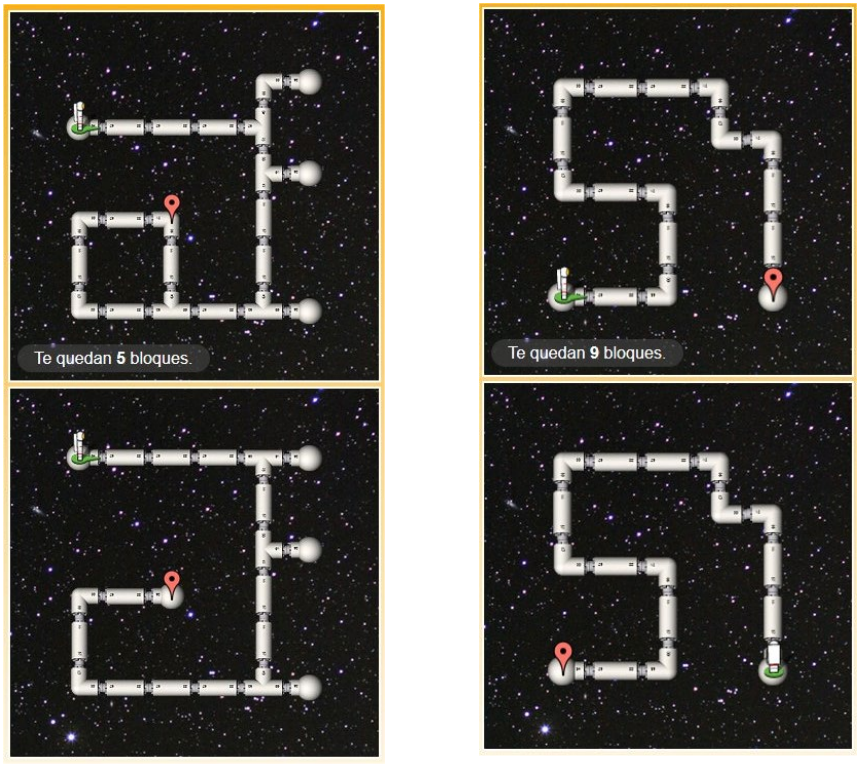


Figura 5.6. Laberintos de niveles 12 (izda.) y 13 (dcha.) de KodetuGen2.

A partir del nivel 12 (figura 5.6 izquierda), se mantiene el diseño de KodetuGen, ya que pretendemos utilizar los últimos niveles como referencia para analizar las diferencias de aprendizaje en función de los cambios realizados en la plataforma. El nivel 12 refuerza la generalización de repetitivas con alternativas, con un laberinto que permite varios caminos, con límite de 5 bloques. El nivel 13 (figura 5.6 derecha) incorpora las alternativas dobles y obliga a mayor anidamiento de estructuras, viéndose que el mismo laberinto se resuelve al revés. Permite además mucha expresividad de código con límite de 10 bloques. Soluciones posibles:

Nvel 12: `[rep [ad si-d [gd]]]`

Nivel 13: `[rep [si-ad [ad] si-no [si-i [gi] si-no [gd]]]]`

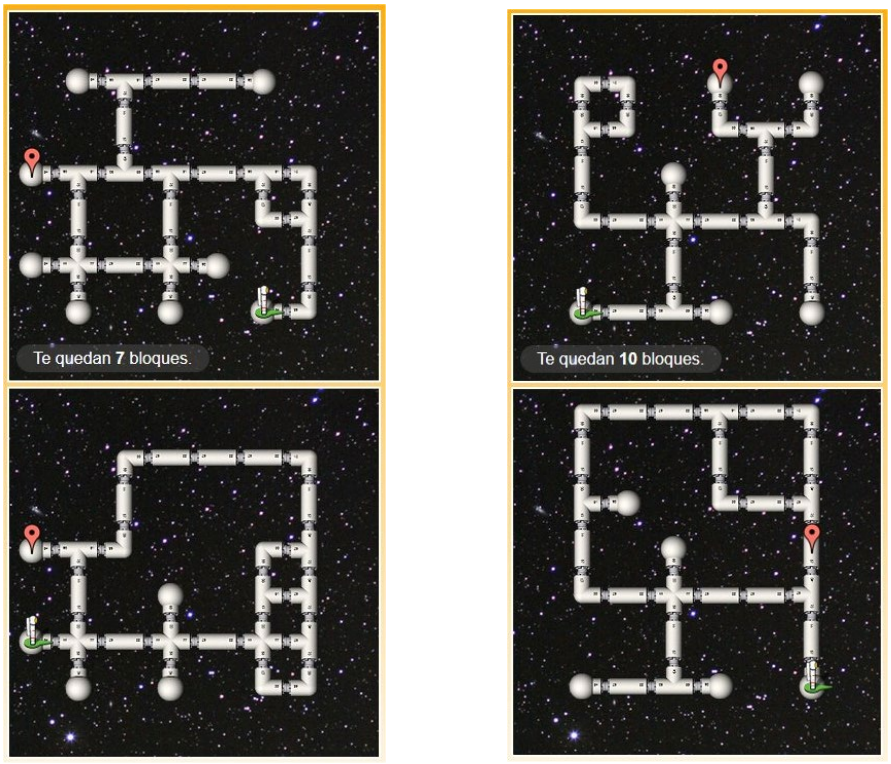


Figura 5.7. Laberintos de niveles 14 (izda.) y 15 (dcha.) de KodetuGen2.

El nivel 14 (figura 5.7 izquierda) es una generalización con un laberinto más complejo, mientras que el nivel 15 tiene el laberinto más complicado, con un bucle que obliga a diseñar muy bien el orden de comprobación de las alternativas para que el astronauta no se pierda en un bucle infinito (figura 5.7 derecha). Los límites de bloques son, respectivamente, 7 y 10. Algunas soluciones posibles:

Nivel 14: `[rep [si-ad [ad] si-no [si-i [gi] si-no [gd]]]]`

Nivel 15: `[rep [si-i [gi ad] si-no [si-ad [ad] si-no [gd]]]`

El resto de la interfaz, aparte de modificaciones estéticas sencillas, se mantiene como en Kodetu, con los laberintos y el botón de ejecución a la izquierda, una zona de bloques disponibles intermedia y un espacio de trabajo a la derecha ocupando

la zona principal de pantalla. Desaparecen las opciones de la ejecución de doble velocidad y la ejecución independiente de cada laberinto de KodetuGen (figura 5.8).

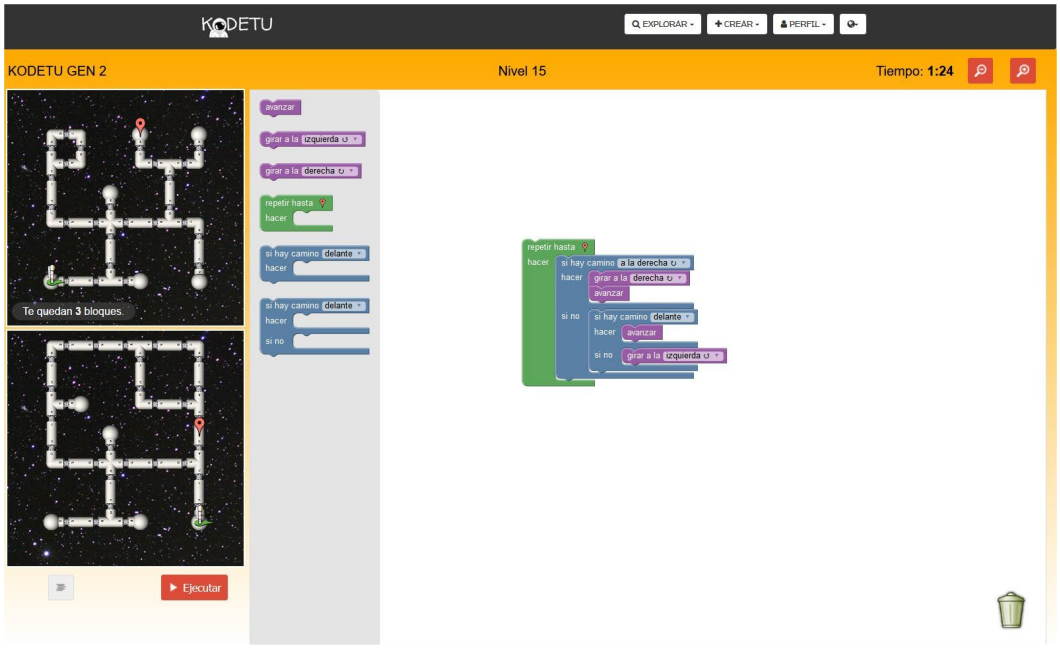


Figura 5.8. Aspecto general del interfaz de KodetuGen2.

5.2.2. Modificaciones en KodetuWin

Nuestra herramienta de gestión de datos de uso de Kodetu necesita modificaciones significativas para adaptarla a esta versión. Aunque el cambio en el diseño de niveles mantiene 15 niveles en total que se siguen codificando como números en el rango 1-15, la comparativa posterior ya no se podrá realizar estrictamente nivel a nivel. Por ello, incorporamos a los datos básicos una nueva codificación de niveles, como se observa en la tabla 5.1. Estos códigos nos permitirán comparar aquellos niveles que son equivalentes en cuanto a planteamiento, aunque no estén en el mismo orden dentro de su versión de Kodetu.

Como se puede observar en la tabla, añadimos además un concepto de **tipo** que hace referencia a los bloques disponibles en cada nivel:

- Tipo A: solo bloques secuenciales (avance y giro: **`ad, gi, gd`**)
- Tipo B: secuencias y repetitivas (se incorpora **`rep`**)
- Tipo C: secuencias, repetitivas y alternativas simples (se incorporan **`si-i, si-d, si-ad`**)
- Tipo D: secuencias, repetitivas y alternativas simples y dobles (con **`si-no`**)

Como comentaremos más adelante, para hacer este análisis por tipo no consideraremos los niveles de entrenamiento (el 1 y el 8 -7 en KodetuGen2-), que aparecen por ello sin tipo en la tabla.

Nivel por versión			Cod. nivel	Tipo	Límites
Kodetu	KodetuGen	KodetuGen2			
1	1	1	K01		no
2	2	2	K02	A	no
3	3	3	K03	A	no
4	4		K04	A	no
		4	K04	A	no
5	5	5	K05	A	no
6	6		K06	A	no
7	7		K07	A	no
		6	K07	A	no
8	8		K08		2
		7	K08		2
9	9		K09	B	5 (6 en KG)
		9	K09	B	8
		10	K09b	B	no
10	10	8	K10	B	5
11	11		K11	C	5
		11	K11	C	8
12	12	12	K12	C	5
13	13	13	K13	D	10
14	14	14	K14	D	7
15	15	15	K15	D	10

Tabla 5.1. Codificación de los niveles de las tres versiones de Kodetu.

Por otra parte, debido al importante número de variables que vamos a calcular de cara a hacer el análisis comparativo, nos planteamos la oportunidad de incorporar un sistema preliminar de visualización de resultados estadísticos, que nos permite durante el proceso de datos probar cálculos de variables y observar de forma interactiva los valores agregados y los estadísticos esenciales (medias, medianas, desviación típica) sobre cualquier conjunto de datos. De esta forma, incorporamos en KodetuWin un subsistema que permite realizar y visualizar este tipo de consultas (figura 5.9).

Otras modificaciones de calado nos obligan a considerar aspectos nuevos que desarrollamos en las próximas secciones: la diferenciación entre sesiones y usuarios, el filtrado de conjuntos de datos y la identificación y marcado de grupos.

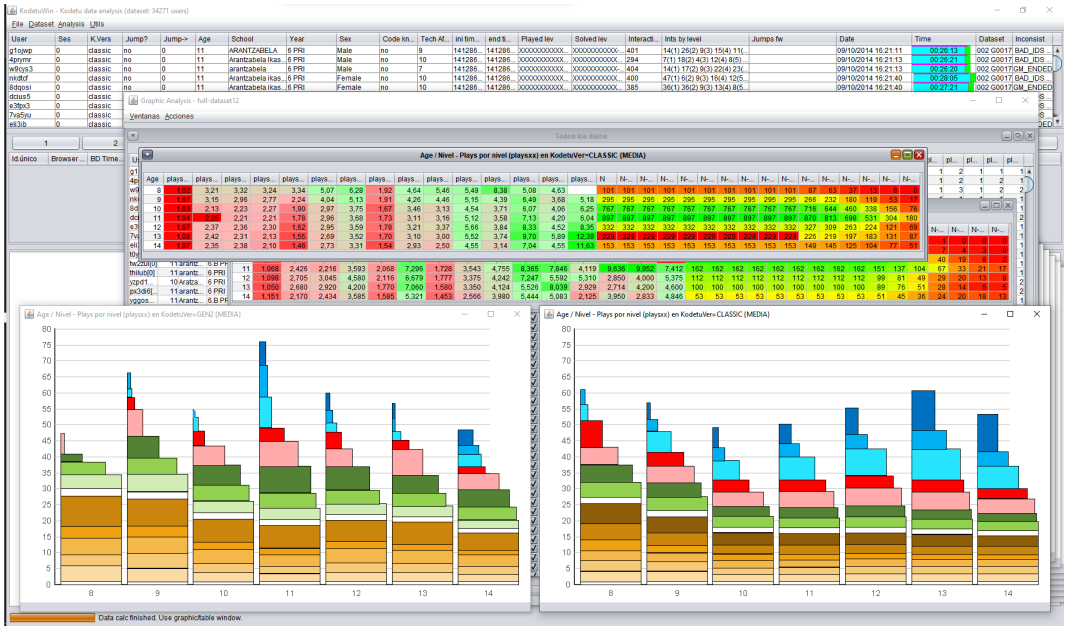


Figura 5.9. Ejemplo del módulo de visualización de KodetuWin.

5.2.3. Diferenciación de usuarios y sesiones

En KodetuGen2, un aprendiz puede ya realizar distintas sesiones en la herramienta al ser identificado con un código de usuario único. Esto nos permitirá localizar más fácilmente aquellos casos en los que el aprendiz vuelve a iniciar el proceso de retos desde el inicio, o entra por segunda vez en la aplicación. Desde el punto de vista de la investigación, desecharemos las sesiones posteriores ya que, obviamente, el entrenamiento previo y el propio recuerdo de los niveles ya afrontados hacen que la situación de partida no sea homogénea.

Por ello, incorporamos en esta versión de KodetuWin el desacoplamiento entre los conceptos de usuario y sesión. A cada usuario ahora le corresponde un código alfanumérico único (como antes), pero puede contener de 0 a n sesiones, que entendemos como todas las interacciones consecutivas que ocurren en el proceso secuencial de la aplicación empezando siempre en el nivel 1 y hasta donde el usuario acaba, abandona o vuelve a iniciar. Para ello, la herramienta de análisis observa cuándo la marca temporal de las interacciones no sigue la lógica de niveles secuencial y diferencia la sesión cada vez que ocurre. También se marca un corte de sesión cuando el tiempo pasado entre dos interacciones es demasiado grande (definimos el tiempo límite entre interacciones en 30 minutos).

Con todo ello, rediseñamos el código único de sesión con una combinación del código único de usuario seguido del número de sesión (secuencial, empezando en cero) entre corchetes. Por ejemplo, el usuario *g1ojwp* tiene una primera sesión con el código único *g1ojwp[0]*. Todas las sesiones [1], [2], etc. se desecharán para la investigación principal.

5.2.4. Despliegue del experimento

La plataforma KodetuGen2 se encuentra activa desde marzo de 2018. Desde ese momento hasta el cierre de la captura de datos para esta investigación en enero de 2019, han accedido 3.233 usuarios que han realizado 3.256 sesiones, en las cuales se han registrado 2.148.036 interacciones.

Una parte de esas sesiones son usuarios no controlados, que acceden a través de Internet desde cualquier lugar del mundo. En paralelo, organizamos **sesiones controladas** con centros escolares de primaria y secundaria, junto con unos pocos grupos de bachillerato y aprendices universitarios. En estas sesiones, realizadas en aula con un ordenador por persona, se presenta el experimento de manera conjunta, se pide a los aprendices que lo resuelvan sin ayuda y se les deja realizar el reto de KodetuGen2. La duración habitual de estas sesiones es de 50 minutos, aunque como comentaremos más adelante, resulta complicado que el tiempo de experimentación sea siempre el mismo.

Los grupos de escolares que participan en el experimento son de dos tipos: 1) concertados previamente (en cuyo caso se preparan y precodifican para que antes de empezar se les entregue un código, generado aleatoriamente, que les permite entrar en la plataforma con el grupo ya identificado) y 2) centros que no se pueden preparar con antelación (y en ese caso se registra la información de la sesión y se identifican los usuarios posteriormente).

5.2.5. Filtrado y validación de sesiones e interacciones

Tras cerrar el periodo experimental, realizamos el proceso de captura y filtrado de toda la información registrada en nuestro servidor. La idea de este proceso de bajo nivel sobre conjuntos de datos está identificada en la literatura en procesos de analítica combinada, como hacen Grover y cols. (2017a), en nuestro caso lo hemos integrado con el resto de utilidades de KodetuWin. Realizamos las siguientes acciones de filtrado y comprobación sobre la información original:

1. Revisión de corrección de éxito. Se recorren todas las interacciones con marca de éxito de nivel y se comprueba que el código registrado en esas interacciones resuelve efectivamente los laberintos correspondientes. Se desechan las sesiones completas donde hay algún error (debido a registros incompletos de la sesión).
2. Revisión de saltos de nivel. Se comprueba que la secuencia temporal de interacciones de cada aprendiz tiene el nivel correcto (cada interacción está en el mismo nivel o el siguiente a la anterior y si hay cambio de nivel debe haber antes una solución exitosa del anterior). Algunos aprendices operan con el navegador para volver a niveles anteriores y también hay errores de comunicación que hacen que se pierdan interacciones. Las sesiones donde ocurren estos errores se desechan.

3. Comprobación de sesiones consistentes. Se revisan todas las sesiones comprobando si no hay pausas excesivas (más de 30 minutos entre una interacción y la siguiente), si son demasiado cortas (menos de 10 interacciones o no se pasa del nivel 2) o usuarios sin datos suficientes de registro. Si se da cualquiera de estas circunstancias, estas sesiones se desechan.
4. Cálculo de grupos controlados. Para ello se utilizan los códigos de grupo ya introducidos y además se diseña un algoritmo que identifica los bloques de usuarios que han entrado en la plataforma de manera concurrente en función de las horas de inicio y final de sesión y del nombre de centro escolar. Se marcan como posibles grupos y luego se contrasta manualmente la información de edad y centro para comprobar que el grupo es correcto. En este punto se diferencian el conjunto de datos de grupos controlados y el resto, desechándose estos últimos de cara al análisis posterior. Se codifica en cualquier caso cada aprendiz con un número de grupo único, que identifica al grupo de escolares que realiza la prueba de forma conjunta.
5. Se eliminan las sesiones de edades menores que 6 o mayores que 21.

Al realizar la revisión de interacciones, se observan algunos casos extraños (interacciones no naturales en cuanto a la secuencia) que nos determinan a hacer validaciones adicionales sobre las sesiones filtradas. En particular, consideramos que las secuencias de código tienen que ser siempre lógicas al examinarlas de forma secuencial (de acuerdo a la marca temporal) ya que cada interacción guardada en el registro es almacenada cuando el aprendiz realiza alguna interacción en la plataforma y almacena la marca temporal del navegador y el estado de código del espacio de trabajo en ese momento. Es decir, estas interacciones deben corresponder a un cambio de nivel, un bloque añadido, una modificación de bloque (en los que lo permiten), un conjunto único de bloques eliminados o un movimiento de un conjunto único de bloques de un lugar a otro del código. Denominamos a esto **cambio atómico** y desarrollamos un sistema que revisa cualquier pareja de interacciones consecutivas de Kodetu y determina los cambios realizados entre ambas, así como la información de si el cambio es atómico o no. Este análisis de modificaciones progresivas de código se propone en algunos estudios como el de Blikstein y cols. (2014).

Se diseña y desarrolla entonces un nuevo algoritmo que realiza esta comprobación de cambios atómicos recorriendo en orden temporal todas las secuencias de interacciones dentro de cada nivel comprobando los cambios de nivel y código. Se encuentran las siguientes incidencias:

- Faltan interacciones. En algunos casos, probablemente por fallos de comunicación puntuales, hay interacciones que responden a varios cambios atómicos en lugar de a uno. Se revisa una muestra significativa de estos casos y se determina que la mayor parte de las veces son dos inserciones seguidas de bloques donde se pierde la primera inserción y se registra solo la segunda. También hay casos de dos eliminaciones y de dos modificaciones (de condiciones de alternativas o giros izquierda/derecha). Como en el análisis posterior se medirá el número de cambios de código y no hay diferencia en este caso en tener o no una interacción intermedia, se mantienen estos casos.

  Hay unos pocos casos en los que los cambios son modificaciones de otro tipo, o hay más de dos cambios atómicos que no se pueden inducir de forma unívoca: se desechan esas sesiones.
- Hay interacciones mal temporizadas. Unos pocos casos de interacciones no tienen transiciones de código lógicas y su revisión determina que se han registrado mal temporizadas. Determinamos que se debe a la manera en la que el código JavaScript en el navegador web procesa de forma asíncrona el almacenamiento de los registros. Vemos que esos casos se pueden identificar porque el número secuencial almacenado y el tiempo de registro en la base de datos no coincide con el orden determinado por la marca temporal. En ese caso desarrollamos una heurística para recolocar la interacción en su lugar correcto. Si es posible se incluye; y si no lo es, se desechan las sesiones.
- Reinicio no registrado. Desde el inicio de nuestro análisis con Kodetu registramos todas las interacciones de cada aprendiz con el espacio de trabajo y las pulsaciones del botón de ejecución. El botón de reinicio (el que hace que se pueda volver a hacer una ejecución después de una fallida) no se consideró interesante para el registro. Sin embargo, detectamos que en unas pocas ocasiones hay un código correcto con una ejecución asociada que no acaba en un final de nivel. Revisando la secuencia de estas interacciones, concluimos que lo que ocurre en esos casos es que el aprendiz empieza a ejecutar un código y, antes de que acabe, piensa que no va a ser correcto y por ello lo detiene, aunque realmente habría solucionado el laberinto. El sistema de registro en esos casos registra una ejecución exitosa, pero el nivel tiene que seguirse jugando porque no se ha llegado a finalizar. Desarrollamos un proceso que revisa todas las secuencias en busca de estos casos y los marca para poder valorarlos después.

- Cambios tardíos de espacio de trabajo. También detectamos algunos casos en los que el nivel es superado con éxito, pero el código final del espacio de trabajo es diferente al del código correcto que soluciona el nivel. Esto ocurre porque la aplicación permite que se modifiquen los bloques de código mientras el astronauta está moviéndose. Aunque es raro que ocurra, algunos aprendices realizan ese proceso. Decidimos mantener esas interacciones pero no se considerarán en el cálculo de indicadores ya que son posteriores al momento de la ejecución del código correcto.

Tras todas estas comprobaciones, se procede al trabajo particular con los datos de grupos de aprendices.

5.2.6. Identificación de grupos y marcado de sesiones

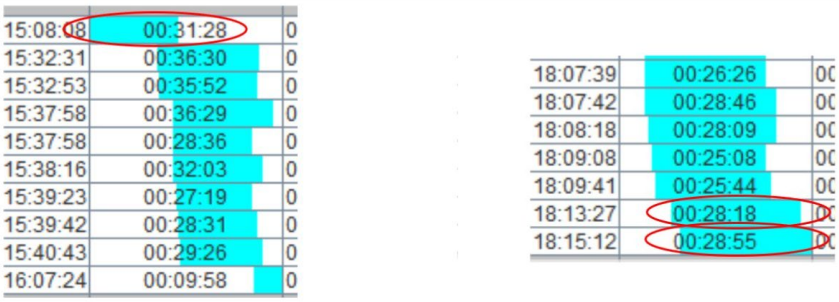
A lo largo de la experimentación, se detectan aprendices que por problemas técnicos o incidencias personales (llegar tarde, no encontrar el aula, ausentarse al baño, tener que irse antes, etc.) no tienen el mismo tiempo de sesión que otros compañeros. Nos planteamos la importancia de discriminar el tiempo de sesión de cada aprendiz dentro de su grupo ya que estas circunstancias pueden afectar de manera significativa al desempeño.

Para ello revisamos todo el conjunto de datos observando las sesiones de forma relativa dentro de cada grupo. Primero, realizamos una observación sobre una muestra significativa de los grupos existentes, donde tenemos ya calculados los datos de hora de inicio y final de cada sujeto, tiempo absoluto de sesión y tiempo con respecto a su grupo. Para la observación, construimos tablas de grupo donde representamos visualmente esta información con un coloreado cian, siendo toda la casilla "Time" la que representa el tiempo total del grupo (desde el primer inicio hasta el último final), como se ve por ejemplo en la tabla 5.2:

Date	Time
13/04/2018 16:13:48	00:36:08
13/04/2018 16:13:48	00:30:46
13/04/2018 16:13:48	00:19:39
13/04/2018 16:13:49	00:37:01

Tabla 5.2. Tiempos de aprendices coloreados de forma relativa.

Dentro de cada grupo, ordenamos los aprendices por tiempo de inicio y hacemos un análisis de las distintas circunstancias temporales que influyen en nuestra investigación. En primer lugar, vemos una característica de desviación temporal que ocurre en algunos grupos en las primeras o últimas sesiones (marcadas con óvalos rojos en la figura 5.10).



15:08:08	00:31:28	0
15:32:31	00:36:30	0
15:32:53	00:35:52	0
15:37:58	00:36:29	0
15:37:58	00:28:36	0
15:38:16	00:32:03	0
15:39:23	00:27:19	0
15:39:42	00:28:31	0
15:40:43	00:29:26	0
16:07:24	00:09:58	0

18:07:39	00:26:26	00
18:07:42	00:28:46	00
18:08:18	00:28:09	00
18:09:08	00:25:08	00
18:09:41	00:25:44	00
18:13:27	00:28:18	00
18:15:12	00:28:55	00

Figura 5.10. Tablas de sesiones con primeras o últimas sesiones del grupo marcadas.

Estos casos se deben en su mayor parte a una diferencia de reloj del equipo en el que se ha desarrollado el experimento. Al no poder considerar que haya sincronización de reloj con el resto del grupo, los tiempos de estos individuos no tienen que considerarse para marcar el tiempo general (inicial o final) del grupo. Podremos incluir a estos sujetos en el análisis y considerar sus marcas temporales relativas, pero no considerar de forma absoluta su marca temporal. Identificamos estas sesiones con una marca de *sesión-desplazada* y no consideramos sus tiempos iniciales o finales en el cálculo de tiempos del grupo (sí en el cálculo de medias). Para considerar a una sesión como desplazada definimos la siguiente heurística:

- El grupo tiene más de 4 sesiones
- La sesión es una de las dos primeras o las tres últimas del grupo
- Su duración es de al menos 5 minutos
- Si es de las primeras, es al menos 5 minutos anterior a la primera válida. Si es de las últimas, empieza al menos 10 minutos después del inicio de la sesión y acaba después del tiempo final estimado de grupo (comentamos esto a continuación).

Marcamos estas sesiones para indicar que son diferentes y no se posicionarán de forma absoluta temporal con respecto al resto, sino en el inicio o en el final, según corresponda.

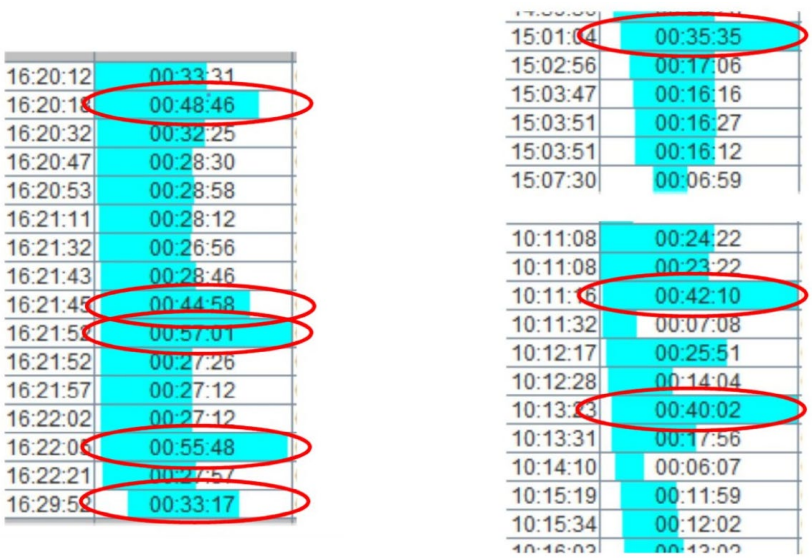


Figura 5.11. Algunas sesiones con aprendices que se extienden más allá de su grupo.

Observamos otra característica que tendremos en cuenta en la figura 5.11. En algunos casos (marcados con elipses rojas) unos pocos aprendices permanecen en la sesión más tiempo que sus compañeros (las barras cianes que se extienden más a la derecha del resto). Marcaremos esas sesiones para poder elegir posteriormente si analizarlas o no. Para ello, calculamos en cada grupo el **tiempo final estimado de grupo**, del siguiente modo:

- Partimos del 70% de las sesiones que primero acaban y tomamos el mayor tiempo final.
- Añadimos todas las sesiones que hayan acabado como máximo 1,5 minutos después de ese.
- El tiempo final de la última de esas sesiones acabada marca el tiempo final estimado de grupo.
- Si el tiempo final de grupo es muy cercano (menor de 1,5 minutos) al tiempo final de la última sesión, se considera ese como tiempo final estimado de grupo.

Representamos en nuestro sistema este tiempo final con una línea vertical azul oscuro que coincidirá en todo el grupo. Las sesiones que se pasan de ese tiempo las marcamos como *sesión-extendida* y coloreamos la parte extendida en amarillo. Se observa un ejemplo de estas sesiones en la figura 5.12.

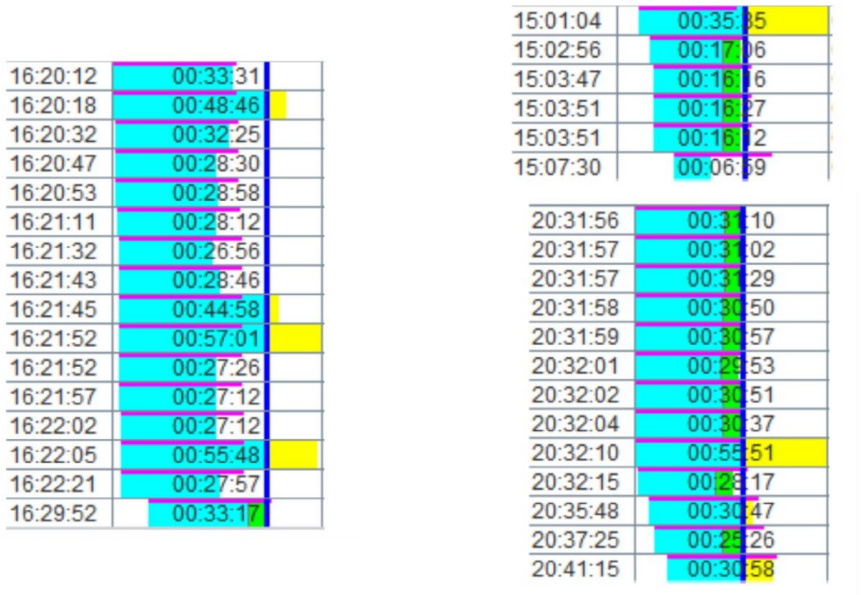


Figura 5.12. Tablas de tiempos de sesión de grupo con líneas de final y sesiones extendidas.

En esta figura hemos añadido también:

- Una banda verde para las sesiones finalizadas en los últimos tres minutos del tiempo final de grupo: *sesión-cerrada*. Esta marca nos servirá para identificar a aprendices que probablemente han cerrado la sesión porque el experimento acababa. El resto abandonan o acaban libremente, o bien han tenido algún problema técnico con la herramienta (cierre del navegador, apagado del equipo, etc.)
- Una línea púrpura de base con el tiempo medio de sesión de cada grupo

Por último, consideramos otras dos circunstancias:

- Hay sesiones que son demasiado cortas, las identificamos con una marca de *sesión-corta* y visualmente la marcamos en rojo.
- Las sesiones que empiezan demasiado tarde en el tiempo del grupo, bien porque el aprendiz se ha incorporado demasiado tarde, bien porque ha reiniciado sesión las identificamos como *sesión-tardía* cuando empieza más de 15 minutos después del inicio del grupo, o cuando empieza solo 5 minutos antes del final estimado (y la marcamos en naranja).

Vemos en la figura 5.13 algunos ejemplos ya con todas las marcas.

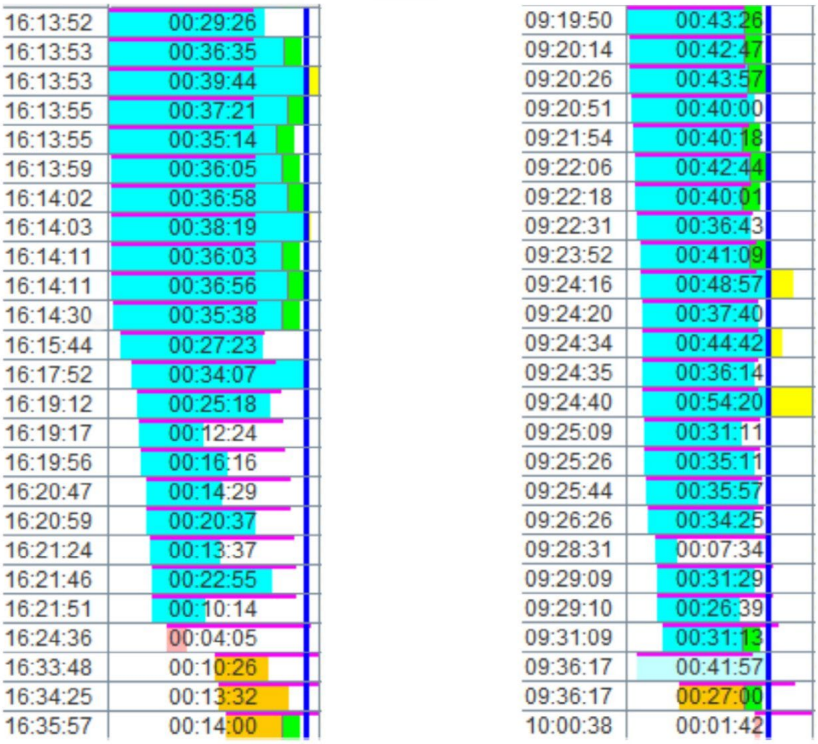


Figura 5.13. Tablas ejemplo de tiempos de sesión de grupo con marcas.

Tras este proceso de marcado, se incorpora en KodetuWin el trabajo automatizado con todas las sesiones y se realiza la eliminación de cara al posterior análisis estadístico de las sesiones cortas, las tardías y las extendidas.

De cara al cálculo de tiempo de sesión, se considera el tiempo estimado de grupo como el tiempo límite de cada aprendiz en su sesión.

5.2.7. Construcción del conjunto de datos definitivo

Con todas estas consideraciones, construimos nuestro tercer conjunto de datos. Aunque haremos análisis de las sesiones de KodetuGen2, también serán muy importantes las comparaciones con los desempeños de los aprendices en Kodetu y KodetuGen. Por ello, hacemos el mismo trabajo de filtrado y validación reseñado con las sesiones registradas anteriormente de Kodetu y KodetuGen.

Los datos de KodetuGen utilizados corresponden a los usuarios analizados en el cuarto capítulo ya que cerramos la web al abrir KodetuGen2. En el caso de Kodetu, sin embargo, al haber seguido manteniendo la plataforma abierta, hemos añadido también aprendices posteriores no incluidos en el conjunto de datos del tercer capítulo, hasta que cerramos la recogida de datos para esta investigación.

En los tres casos, incluimos solo los grupos controlados y desechamos a los usuarios libres para todo el trabajo realizado en este capítulo, haciendo también el mismo filtrado de grupos y sesiones que acabamos de describir.

Este conjunto de datos se compone entonces del siguiente modo:

- Kodetu: 4.079 aprendices de edades entre 6 y 21, con 1.739.930 interacciones, entre 9/10/2014 y 6/2/2018.
- KodetuGen: 647 aprendices entre 9 y 17 años, con 507.730 interacciones, entre 16/9/2016 y 13/3/2018.
- KodetuGen2: 1.263 aprendices entre 7 y 21 años, con 888.314 interacciones, entre 13/4/2018 y 22/1/2019.

En total, 5.989 aprendices de edades entre 6 y 21, con 3.135.974 interacciones, entre 9/10/2014 y 22/1/2019.

De cara a la mayor parte de los análisis, vemos que la muestra de las distintas edades es muy diferente en número y en variedad de centros. Filtramos con ello las muestras por edades y elaboramos un segundo conjunto de datos con las edades más representativas para nuestro estudio, de 8 a 14 años, que es el que utilizaremos en adelante salvo mención explícita. Corresponde a los aprendices de segundo y tercer ciclo de primaria y primer ciclo de secundaria, siendo su composición:

- Kodetu: 3.141 aprendices de edades entre 8 y 14, con 1.415.610 interacciones.
- KodetuGen: 598 aprendices entre 9 y 14 años, con 476.253 interacciones.
- KodetuGen2: 1.004 aprendices entre 8 y 14 años, con 732.478 interacciones.

En total, 4.743 aprendices de edades entre 8 y 14 (edad media 10,8), con 2.624.341 interacciones, entre 9/10/2014 y 22/1/2019.

5.2.8. Métodos estadísticos

Dada la distribución de valores del colectivo estudiado la normalidad de los valores es muy improbable. Aplicamos a todas las variables obtenidas y calculadas de nuestro conjunto de datos la prueba de normalidad de Kolmogorov-Smirnov. En todos los casos el p-valor es muy inferior a 0,001, con lo que efectivamente rechazamos esta opción y la posibilidad de utilizar la prueba t de Student o cualquier otro test paramétrico.

Por ello utilizaremos las pruebas no paramétricas de Mann–Whitney–Wilcoxon (M-W-W) para comparar dos muestras y la de Kruskal–Wallis (K-W) comparando tres o más.

Por otra parte, y debido a la falta de normalidad de las variables utilizadas, en los indicadores donde tenga más sentido, utilizaremos medianas en vez de medias, particularmente en los relacionados con el tiempo, que al estar medido en milisegundos tiene un espacio continuo de variación.

5.3. Resultados

En esta sección, analizamos los resultados específicos de los aprendices seleccionados que utilizaron KodetuGen2. En el siguiente capítulo, haremos la comparación con las otras dos versiones de Kodetu.

5.3.1. Tiempo de juego

Para observar la importancia del tiempo de juego, vemos en la tabla 5.3 la relación directa que hay entre el tiempo de la sesión de juego (en intervalos de 5 minutos) y el nivel finalmente superado. Cuanto más tiempo se juega a KodetuGen2, más se progresa, de una forma prácticamente lineal.

		Nivel conseguido (% sobre inicial)														
	N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0-10 mins	60	100%	100%	85%	62%	45%	12%	8%	2%	0%	0%	0%	0%	0%	0%	0%
11-15 mins	57	100%	100%	96%	88%	81%	47%	35%	26%	19%	9%	2%	0%	0%	0%	0%
16-20 mins	70	100%	100%	97%	89%	86%	63%	56%	44%	40%	24%	16%	1%	1%	1%	1%
21-25 mins	102	100%	100%	95%	91%	84%	70%	65%	58%	41%	23%	10%	3%	2%	2%	1%
26-30 mins	210	100%	100%	99%	97%	93%	79%	74%	60%	45%	28%	12%	8%	4%	3%	2%
31-35 mins	190	100%	100%	99%	99%	98%	89%	84%	79%	66%	38%	17%	10%	8%	6%	4%
36-40 mins	127	100%	100%	98%	98%	97%	90%	89%	82%	67%	48%	24%	16%	11%	7%	2%
41-45 mins	96	100%	100%	100%	100%	99%	98%	97%	92%	88%	65%	35%	17%	9%	5%	2%
46-50 mins	38	100%	100%	100%	95%	95%	89%	89%	87%	79%	66%	37%	18%	8%	0%	0%
51-55 mins	15	100%	100%	93%	93%	93%	93%	87%	87%	87%	80%	53%	33%	13%	7%	7%
56-65 mins	25	100%	100%	96%	96%	96%	96%	96%	96%	96%	84%	76%	60%	28%	8%	4%
66-100 mins	14	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	86%	71%	64%	50%	0%

Tabla 5.3. Mapa de calor del nivel conseguido en función del tiempo de juego.

Aunque la mayor parte de nuestros aprendices se encuentran en la franja de 21 a 45 minutos, en esta tabla se observa que jugar a KodetuGen2 puede requerir más tiempo del que definimos como objetivo en el experimento: la complicación adicional que significa la genericidad necesita mayor dedicación que la utilizada en todo este estudio. Para lograr un uso óptimo de la herramienta en las edades observadas, sería recomendable dedicar sesiones algo superiores a una hora y no los 50 minutos objetivo que hemos mantenido en este experimento (veremos en la comparativa que esto no ocurre en Kodetu). El nivel medio alcanzado progresa linealmente (figura 5.14) con respecto al tiempo (K-W: chi-cuadrado = 349,57, df = 13, p-valor < 0,001), aunque no podemos asegurar cuál sería el tiempo objetivo ideal al no haber realizado sesiones de tiempo superior en las que se observe que un incremento de tiempo disponible no redunda en un resultado equivalente.

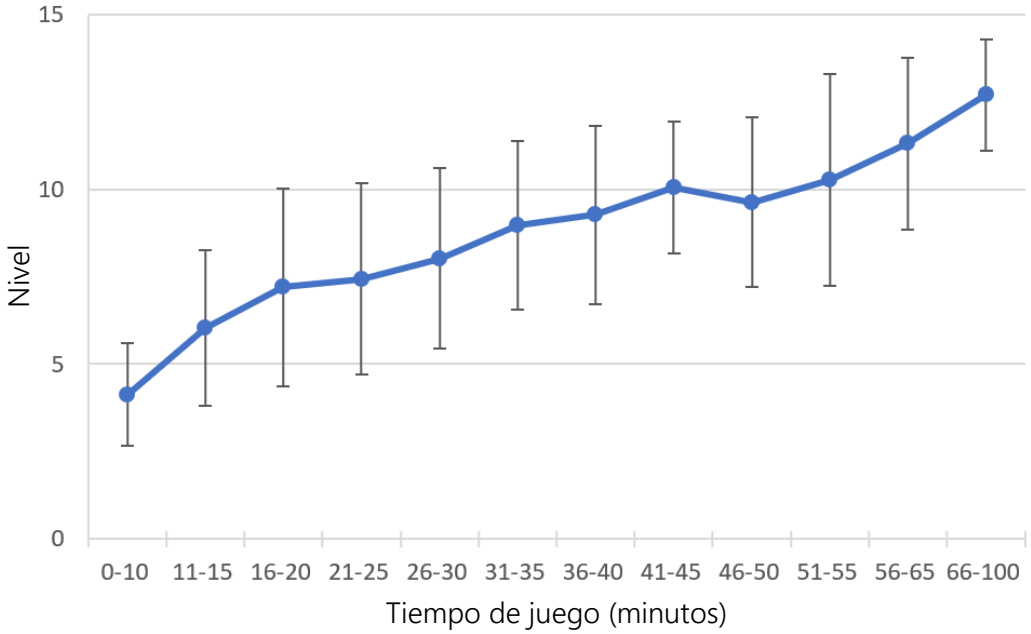


Figura 5.14. Evolución del nivel medio superado con respecto al tiempo de juego (intervalos).

5.3.2. Resultados en función de la edad

Superación de niveles

La dificultad creciente de los niveles y la limitación temporal causa la progresión descendente en la superación de niveles desde el nivel 1 al nivel 15. Si segmentamos por edad, se puede observar cómo esta influye en la progresión (tabla 5.4), en función del porcentaje de aprendices que superan cada nivel con respecto a los iniciales. En nuestro conjunto de datos, todos los aprendices superan los dos primeros niveles (consecuencia lógica de haber eliminado del conjunto de datos las sesiones muy cortas). Después, el porcentaje va disminuyendo hasta solo el 2,2% que supera el nivel 15.

Edad	N (ini1)	Nivel															N (f15)	nSup medio
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
8	70	100,0%	100,0%	91,4%	81,4%	71,4%	48,6%	48,6%	35,7%	18,6%	2,9%	1,4%	0,0%				0	6,0
9	96	100,0%	100,0%	97,9%	92,7%	85,4%	65,6%	58,3%	46,9%	31,3%	14,6%	7,3%	4,2%	3,1%	0,0%		0	7,1
10	292	100,0%	100,0%	99,3%	96,9%	93,5%	80,1%	73,6%	67,5%	57,2%	38,4%	13,7%	6,8%	2,7%	0,7%	0,0%	0	8,3
11	184	100,0%	100,0%	99,5%	98,9%	97,8%	92,9%	88,6%	82,1%	74,5%	57,1%	36,4%	17,9%	12,5%	9,2%	5,4%	10	9,7
12	157	100,0%	100,0%	93,0%	90,4%	87,3%	74,5%	72,0%	63,1%	52,2%	31,8%	18,5%	12,7%	8,3%	5,1%	3,2%	5	8,1
13	135	100,0%	100,0%	97,8%	91,9%	91,1%	77,0%	74,8%	65,9%	56,3%	37,8%	22,2%	11,1%	3,7%	3,7%	2,2%	3	8,4
14	70	100,0%	100,0%	98,6%	94,3%	87,1%	77,1%	77,1%	72,9%	64,3%	51,4%	34,3%	28,6%	25,7%	18,6%	5,7%	4	9,4
Total	1004	100,0%	100,0%	97,4%	93,9%	90,2%	77,4%	73,3%	65,4%	54,8%	36,9%	19,7%	11,2%	7,0%	4,5%	2,2%	22	8,3

Tabla 5.4. Porcentaje de aprendices que superan cada nivel, por cada edad y total
A izquierda y derecha las N inicial y final, en gradiente azul (derecha) el nivel superado medio.

Vemos como la edad influye en el desempeño en el juego: en términos generales, a mayor edad, mejor desempeño. Sin embargo, este crecimiento no es lineal. Hay un

comportamiento especialmente bueno a los 11 años (nivel superado medio 9,7). La figura 5.15 muestra la comparación gráfica de los niveles superados acumulados por edad, donde se observa esa tendencia creciente. Usamos unos códigos de color en esta figura que repetiremos en más ocasiones: tonos anaranjados para los niveles de tipo A (secuencias), tonos verdes para los de tipo B (repetitivas), rojos para los de tipo C (alternativas simples) y azules para los de tipo D (alternativas dobles). Los bloques blancos corresponden a los niveles de entrenamiento (en KodetuGen2, el nivel 1 y el 7).

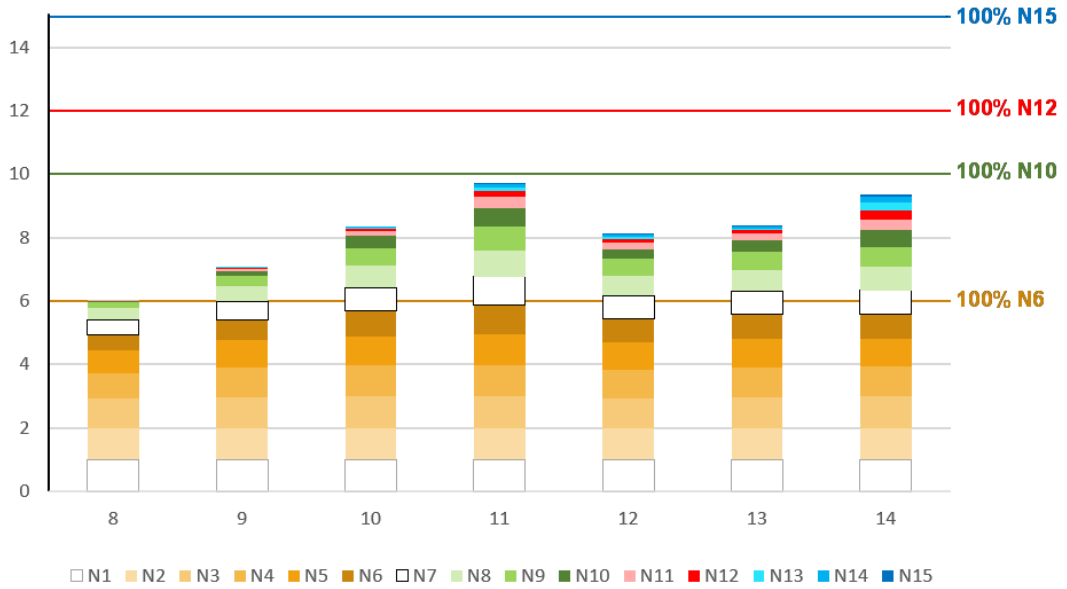


Figura 5.15. Porcentaje acumulado (eje Y) de aprendices que superan los niveles por edad (eje X).

Una posible explicación del rendimiento especialmente bueno en torno a los 11 años podría tener que ver con el contexto psicológico que rodea a la edad de paso de primaria a secundaria. Por ello, hemos segmentado la medición por curso escolar en lugar de por edad, obteniendo una distribución que refuerza la diferencia en el salto de ciclo entre educación primaria y secundaria (figura 5.16).

Consideramos que esta diferencia tan sustancial de rendimiento se relaciona con la tipología del aprendiz preadolescente de secundaria, que puede ver este juego infantil y probablemente tiene una motivación mucho menor para superarlo. Las tasas de abandono explican parcialmente la situación: el 86,9% de los aprendices de 6º de primaria acaban los niveles secuenciales, por solo el 82,3% de 1º de secundaria y el 69,0%. de 2º de secundaria.

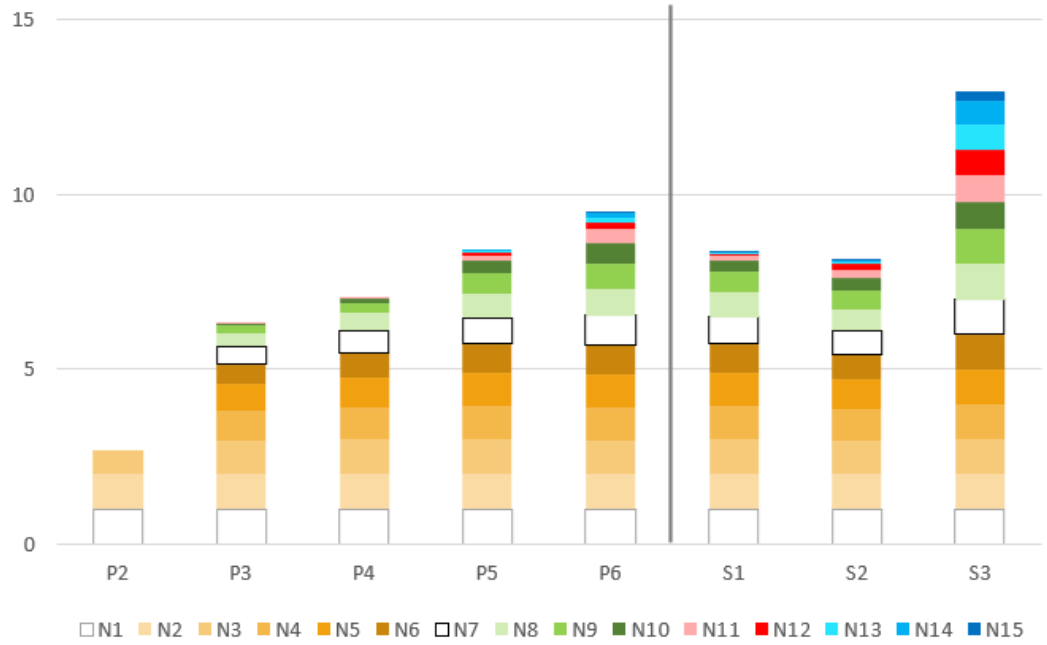


Figura 5.16. Porcentaje de aprendices que superan el nivel 15, por nivel educativo (Pn = curso "n" de primaria, Sn = curso "n" de secundaria).

Para entender esta diferencia, observamos también los tiempos de juego efectivo, calculados en función del tiempo real que cada aprendiz ha estado resolviendo niveles (eliminando los tiempos finales de animación, el tránsito entre niveles y los niveles de entrenamiento 1 y 7). Al observarlo por edades (tabla 5.5) vemos muy claramente que los aprendices de 11 años de edad utilizan de media muchos más minutos efectivos de juego ("T.efectivo") que sus compañeros de mayor edad (30,3 minutos frente a 23,1 o menos).

Edad	T.efectivo	T.sesión	N nivel 1	N nivel 15
8	22,6	29,6	70	0
9	24,7	30,0	96	0
10	25,1	30,5	292	0
11	30,3	35,2	184	10
12	23,1	28,2	157	5
13	20,9	25,1	135	3
14	21,0	25,1	70	4
Total	24,7	29,8	1004	22

Tabla 5.5. Tiempos (minutos) medios de juego efectivo y total de sesión, por edad.

Queda fuera del alcance de esta investigación profundizar en los motivos subyacentes, pero sí nos planteamos la posibilidad futura de modificar el contexto gráfico de KodetuGen2 para ver en qué medida su adaptación a la edad y la diferencia de motivación influyen en esa diferencia de desempeño.

Superación de niveles por tiempo

Otra posible explicación alternativa sería pensar que los aprendices de 11 años tienen un mejor desempeño que los mayores. Con el fin de valorar esta hipótesis, hemos calculado un indicador específico que no está afectado por la motivación o el abandono y que minimiza la importancia del tiempo dedicado: el número de niveles superados por minuto.

Con ese cálculo vemos (figura 5.17) que la supuesta superioridad de los aprendices de 11 años de edad queda descartada. Aquí, ya de forma natural, la progresión de la capacidad de resolver los laberintos de KodetuGen2 es lineal y correlaciona con la edad. Esto redunda en la idea anteriormente mencionada: los aprendices de 11 años tienen un mejor desempeño no porque sean mejores, sino porque lo intentan durante más tiempo que aprendices de más edad (a pesar de que rendimiento en términos de niveles superados por tiempo sea menor).

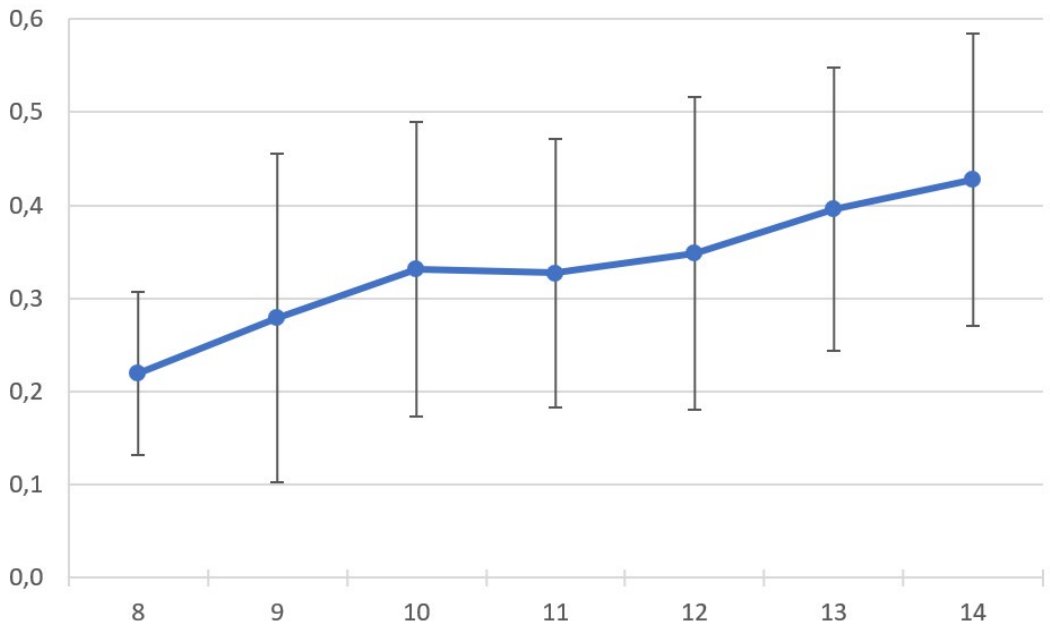


Figura 5.17. Niveles superados por minuto (eje Y), por edad (eje X).

Tiempos por cada nivel

Hemos comentado anteriormente que el tiempo disponible es un elemento clave para el desempeño. Invirtiendo el indicador anterior, hemos calculado el tiempo medio efectivo utilizado por los aprendices en cada nivel. Para analizarlo, hemos diferenciado el éxito del fracaso, de manera que observamos los tiempos en aquellos niveles que se han resuelto exitosamente y de forma independiente los tiempos en los niveles de abandono (solo uno -o ninguno- por cada aprendiz), ya que si integramos ambos estamos contaminando la información de los tiempos de niveles superados con la de los aprendices que fallan en ese nivel.

Realizamos un análisis de los tiempos utilizados por nivel resuelto en cada una de las edades, mostrados en la tabla 5.6. En ella se observan los tiempos promedio que los aprendices de cada edad invierten en cada uno de los 15 niveles (en minutos) cuando son superados con éxito, coloreados en gradiente desde verde (tiempos más cortos) a rojo (tiempos más largos). Hemos añadido también columnas de tamaño de muestra N en el primer nivel y en el último (cada promedio está calculado solo para los aprendices que llegan a ese nivel). La columna de la derecha, con celdas coloreadas en gradiente azul de más oscuro (valores mayores) a más claro (valores menores), corresponde a los minutos totales que cada aprendiz invertiría si acabara el juego con los tiempos medios de cada nivel en su edad (reforzando de este modo nuestra sugerencia de que KodetuGen2 se realice con más tiempo disponible, ya que vemos que el tiempo efectivo medio de los mejores aprendices se acerca o supera los 50 minutos, especialmente en primaria).

Edad	N in1	Nivel															N in15	T. medio (suma)
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
8	70	4,4	3,3	3,7	3,8	2,4	7,8	2,0	2,9	3,1	4,6	10,7					0	48,7
9	96	2,9	2,2	3,3	3,4	2,7	7,2	1,5	3,2	3,7	3,7	10,3	4,0	2,1			0	50,1
10	292	2,4	1,4	2,2	2,3	1,6	6,3	1,6	2,5	4,1	4,6	6,8	4,7	6,3	2,6		2	49,3
11	184	2,2	1,2	1,6	1,8	1,1	5,5	1,0	2,1	3,5	5,4	6,4	3,3	7,5	6,4	3,2	17	52,1
12	157	2,6	1,4	2,0	2,3	1,2	5,2	1,0	1,8	3,0	4,6	5,8	3,9	5,6	5,3	4,6	8	50,5
13	135	1,9	1,0	1,7	2,1	1,0	5,1	1,0	2,2	2,6	3,3	8,4	2,3	3,6	4,3	5,6	5	46,3
14	70	1,7	0,8	1,5	1,9	0,8	3,9	0,8	1,4	2,1	3,4	4,0	1,5	2,6	2,6	4,0	13	33,1
Total	1004	2,5	1,5	2,1	2,4	1,4	5,8	1,3	2,2	3,4	4,5	6,6	3,2	5,2	4,7	4,0	45	50,7

Tabla 5.6. Tiempos (en minutos) utilizados por los aprendices en cada nivel resuelto, por edad.

Podemos observar que los aprendices de menor edad no llegan siquiera a jugar los niveles superiores (ningún aprendiz de 8 años llega a completar con éxito el nivel 12). También se observa la tendencia decreciente natural de tiempo por edad en cada nivel. Se observan algunas particularidades de la evolución de los niveles que ayudarían a un rediseño pedagógico de la herramienta, como el mayor tiempo relativo que hay que dedicar al nivel 6 (el que plantea recorridos elaborados solo con secuencia de movimientos) o la dificultad relativa del nivel 11 (introducción de estructuras alternativas). Por otra parte, observamos el comportamiento del colectivo de 11 años de edad en esta tabla: superan los niveles 13 y 14 a costa de un gran sobresfuerzo convirtiéndose en los aprendices que más tiempo dedican en los niveles de tipo D (alternativas dobles en niveles 13-15). Estas diferencias son significativas para tiempos de nivel 1 a 12 (K-W: p-valor entre 0,02134 y < 0,001), tendencias no significativas en los niveles 13 a 15 (K-W: p-valor > 0,09634).

Aprovechamos esta tabla para introducir un tipo de gráfico que hemos diseñado para este trabajo y que utilizaremos en más ocasiones. Hemos presentado ya algún gráfico de columnas acumuladas (dado que nos interesa ver cómo se van acumulando los desempeños entre niveles), pero a la vez nos interesa recordar cuál es la supervivencia en cada caso (según se va avanzando, cada vez es menor el número de aprendices). Para ello hemos realizado un gráfico de columnas

recortadas en las que según se progresa hacia arriba, en la anchura horizontal se va reflejando la supervivencia, que empieza en el 100% de anchura en los primeros niveles (abajo) y acaba en la anchura proporcional en los niveles superiores (figura 5.18). De esta forma en el mismo gráfico tenemos la información de la variable observada (altura de cada caja), la progresión por niveles en esa variable (cajas apiladas en vertical) y el porcentaje de población restante que forma ese estadístico (anchura en disminución de cada columna). Evidentemente, la confianza estadística va decreciendo según llegamos a los niveles superiores (donde llegan muchos menos aprendices). El color de las cajas sigue diferenciando los niveles, la tonalidad representando al tipo (A-D) de bloques disponibles en cada caso.

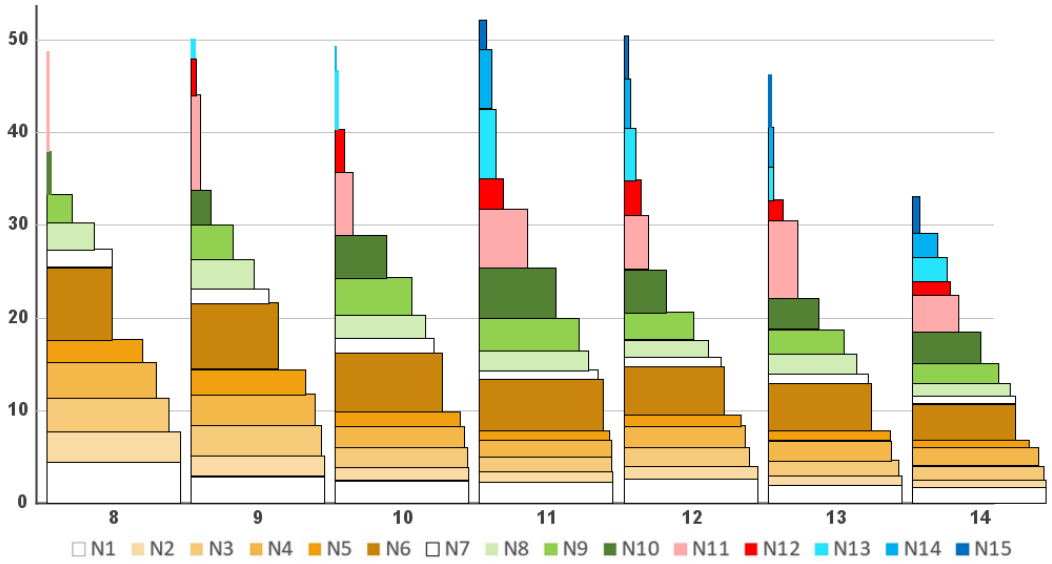


Figura 5.18. Tiempos medios (minutos) utilizados en cada nivel resuelto (eje Y), por edad (eje X).

Podemos observar cómo efectivamente hay diferencias debidas al diferente tiempo disponible en los grupos y cómo han afectado a la segmentación por edad. Los aprendices de 11 años han utilizado más tiempo para KodetuGen2 que el resto de las edades por distintos motivos: 1) más que las edades inferiores porque en ellas no se ha llegado a los niveles más complicados; y 2) más que las edades superiores porque, como comentábamos anteriormente, son los que más se implican en el juego y más aprovechan el tiempo disponible. En el gráfico se observa la tendencia lineal a que los niveles más exigentes (colores verdes, rojos, azules) vayan requiriendo cada vez menos tiempo medio en mayores edades: la progresión natural. En este sentido, podríamos identificar los 14 años como la edad más apropiada de este rango para enfrentarse a los retos de KodetuGen2, aunque todo el rango de 11-14 puede enfrentarse a este aprendizaje con cierta solvencia.

Observando solo los niveles más complicados (los de tipo D), vemos en la figura 5.19 el tiempo invertido en ellos (considerando solo los aprendices que llegan a solucionar el nivel 13 como referencia). Aquí se aprecia cómo 11 parece ser la primera edad adecuada para plantear el reto completo de KodetuGen2. En estos

niveles donde se pone a prueba la capacidad de abstracción algorítmica es donde más se aprecia el desarrollo cognitivo natural asociado a la edad, mientras que en los niveles inferiores (los coloreados en tonos marrón en la figura anterior) el desarrollo no está directamente relacionado con la edad. Parece haber un rango de edades idóneas para cada tipo de reto de PC y la generalización en estructuras de programación compuestas parece relacionarse mejor con el último ciclo de educación primaria y la etapa secundaria.

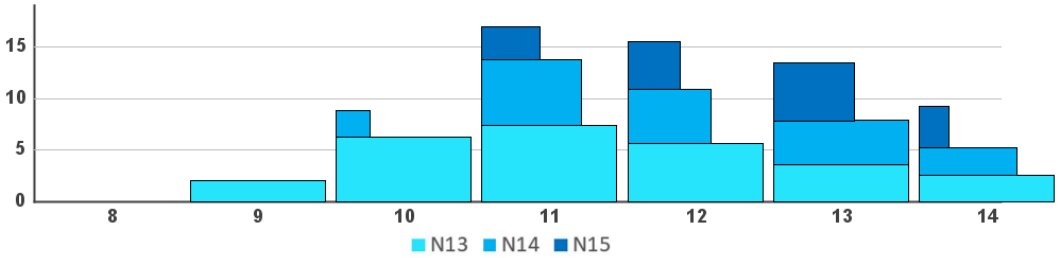


Figura 5.19. Tiempo medio (minutos) invertido en los niveles 13-15 solucionados, por edad.

Abandonos de aprendices

Hemos calculado también los tiempos del nivel no resuelto por los aprendices con el objetivo de completar el análisis de los tiempos por nivel y observar el comportamiento de los abandonos, tal y como se muestra en la tabla 5.7.

Edad	N i1	Nivel															N i15	T.medio/ niv aban.
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
8	70			9,0	3,8	4,9	7,7		2,7	4,6	2,8	0,1	2,6				0	4,2
9	96			0,9	6,6	3,0	6,7	0,6	4,5	3,4	4,8	4,8	2,3	3,4	4,1		0	3,8
10	292			1,0	1,7	3,3	6,9	1,3	5,1	4,3	4,9	5,8	5,5	3,6	3,1	2,5	2	3,8
11	184			1,8	7,3	3,8	5,2	0,9	6,7	5,9	6,5	7,6	7,1	9,3	9,9	6,9	17	6,1
12	157			9,5	3,5	6,0	5,9		4,6	5,4	6,3	3,6	9,7	0,4	2,4	5,1	8	5,2
13	135			6,0	2,2	2,5	5,1	2,9	3,2	4,6	4,2	4,6	1,6	2,2		8,2	5	4,0
14	70			5,4	5,4	2,4	5,0		5,1	3,5	8,2	4,5	1,5	1,3	3,0	5,8	13	4,2
Total	1004			7,2	3,7	3,8	6,3	1,2	4,6	4,6	5,4	5,6	5,8	4,1	4,3	6,1	45	4,8
N aban	927	0	0	26	33	28	128	34	72	107	175	163	80	38	21	22		

Tabla 5.7. Tiempo medio (minutos) de nivel no resuelto por edad y nivel.

Como podemos observar, hay celdas vacías que corresponden a los niveles en los que no hay ningún aprendiz de esa edad que abandone en ese nivel. En la columna de gradiente azul de la derecha, observamos como los aprendices de 11 años son también los que más trabajan antes de abandonar, con un tiempo medio de 6,1 minutos en el nivel que no superan. Son representativos sus tiempos en los niveles 13 o 14, en los que los aprendices están casi 10 minutos intentando resolver el nivel. Analizando los tiempos generales, se ve que es mayor el tiempo de fracaso en el nivel 6 (el más trabajoso de los secuenciales, 6,3 minutos), que en cualquiera del resto de niveles más abstractos.

Asociada a la información de tiempos, podemos observar la tasa de abandono en la tabla 5.8. En cada casilla, indicamos por edad y nivel el porcentaje de aprendices (con respecto al total inicial de su edad) que abandonan el juego en ese

nivel. La gradación de color pasa desde el verde que indica la ausencia de abandono (0%) hasta el rojo que indica los mayores abandonos (20-25%). La columna a la derecha de las celdas coloreadas ("Éxito") marca el porcentaje de aprendices que concluyen exitosamente el nivel 15, es decir, los que no abandonan. La columna derecha es el nivel medio de abandono, formado con la media aritmética ponderada de los abandonos en cada nivel.

En la fila de totales observamos los niveles que marcan los puntos clave de abandono del juego: el 6 (último de los secuenciales) y los niveles 10 y 11 donde la abstracción necesaria para las repeticiones de los dos laberintos hace más complicada la solución.

Edad	N i1	Nivel														Éxito	Niv medio abandono
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
8	70	0,0%	0,0%	8,6%	10,0%	10,0%	22,9%	0,0%	12,9%	17,1%	15,7%	1,4%	1,4%				0,0%	7,0
9	96	0,0%	0,0%	2,1%	5,2%	7,3%	19,8%	7,3%	11,5%	15,6%	16,7%	7,3%	3,1%	1,0%	3,1%		0,0%	8,1
10	292	0,0%	0,0%	0,7%	2,4%	3,4%	13,4%	6,5%	6,2%	10,3%	18,8%	24,7%	6,8%	4,1%	2,1%	0,7%	0,0%	9,3
11	184	0,0%	0,0%	0,5%	0,5%	1,1%	4,9%	4,3%	6,5%	7,6%	17,4%	20,7%	18,5%	5,4%	3,3%	3,8%	5,4%	10,7
12	157	0,0%	0,0%	7,0%	2,5%	3,2%	12,7%	2,5%	8,9%	10,8%	20,4%	13,4%	5,7%	4,5%	3,2%	1,9%	3,2%	9,1
13	135	0,0%	0,0%	2,2%	5,9%	0,7%	14,1%	2,2%	8,9%	9,6%	18,5%	15,6%	11,1%	7,4%	0,0%	1,5%	2,2%	9,4
14	70	0,0%	0,0%	1,4%	4,3%	7,1%	10,0%	0,0%	4,3%	8,6%	12,9%	17,1%	5,7%	2,9%	7,1%	12,9%	5,7%	10,4
Total	1004	0,0%	0,0%	2,6%	3,5%	3,7%	12,8%	4,1%	7,9%	10,7%	17,9%	17,1%	8,6%	4,2%	2,5%	2,3%	2,2%	9,3

Tabla 5.8. Porcentajes de abandono por edad y nivel.

Número de ejecuciones

Hemos analizado también el número de ejecuciones que hacen los aprendices (tabla 5.9) del mismo modo que con las anteriores versiones de Kodetu. Siendo el número ideal 1 ejecución por nivel conseguido, el aumento de este número refleja la complejidad del nivel (bien porque los aprendices necesitan probar cómo va funcionando su programa, bien porque se equivocan al pretender resolverlo). También identifica las estrategias mentales de los aprendices (los que prueban más hacen más ejecuciones, los que son más analíticos piensan mucho antes de cada ejecución). En la tabla se indica el número promedio de ejecuciones por nivel y edad para aquellos aprendices que han superado cada nivel, coloreados de verde (menores) a rojo (mayores). En la fila inferior se muestra el promedio de ejecuciones considerando todos los aprendices y en la columna derecha el número medio de ejecuciones por cada edad (coloreado en tonos de azul, más oscuro cuanto mayor).

Edad	N ini1	Nivel															N fin15	Nº medio de plays
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
8	70	1,1	5,1	3,9	6,2	3,9	9,5	2,3	4,6	4,4	5,5	13,0					0	5,4
9	96	1,0	4,2	4,6	6,2	4,8	9,5	2,2	5,0	5,4	5,5	13,4	4,8	1,7			0	5,3
10	292	1,0	2,9	3,1	4,4	2,6	7,3	2,0	3,5	5,1	6,9	7,7	5,0	4,6	2,5		0	4,2
11	184	1,1	2,7	2,4	4,0	2,2	7,6	1,7	3,2	4,8	8,4	8,2	4,0	10,9	8,1	5,2	10	5,0
12	157	1,1	2,9	3,2	4,6	2,2	6,8	1,8	2,9	4,1	6,7	6,7	3,9	4,2	5,4	4,6	5	4,1
13	135	1,1	3,0	2,9	4,4	1,8	6,9	1,6	3,2	4,2	5,5	9,0	3,9	3,8	4,2	2,0	3	3,8
14	70	1,1	2,4	2,5	3,9	1,6	5,3	1,4	2,4	3,8	4,9	5,0	2,5	4,2	2,7	6,3	4	3,3
Todas	1004	1,1	3,1	3,1	4,6	2,5	7,4	1,8	3,3	4,6	6,8	7,8	3,9	6,3	5,4	4,8	22	4,4

Tabla 5.9. Número de ejecuciones por edad y nivel superado.

Este indicador tiene correlaciones lógicas con los tiempos por nivel (cuantas más ejecuciones se intenten, más tiempo se tarda en ese nivel, tanto se resuelva como se falle). Las dificultades relativas más importantes se encuentran de nuevo en los niveles 6 (por la longitud del camino secuencial y la facilidad con la que el aprendiz se puede equivocar en el programa) y la incorporación de estructuras de control con mayor abstracción de los niveles 9 y sucesivos. Podemos ver en la figura 5.20 cómo la relación con la edad en este indicador es directa, de modo que cuanto mayor es el aprendiz, menos pulsaciones de ejecución necesita para superar cada nivel, encontrándose el mejor comportamiento en la edad 14 (K-W: p-valor<0,05 solo para los niveles 1 al 8).

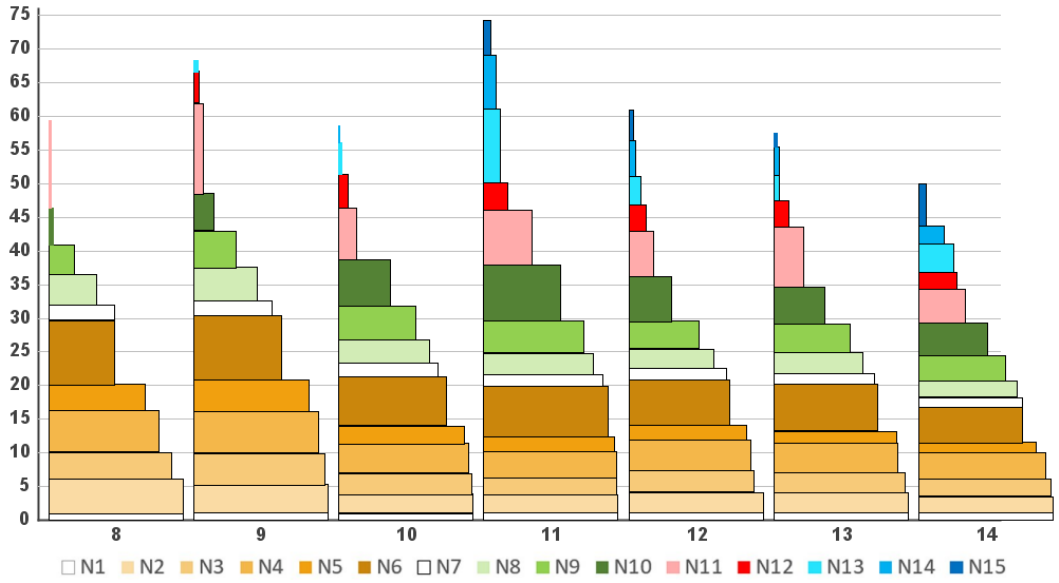


Figura 5.20. Número medio de ejecuciones por edad en niveles superados.

Número de cambios

Acabamos el análisis por edad observando el número de cambios que los aprendices realizan en su código, desde el estado inicial hasta superar cada nivel. Para este análisis tampoco consideramos los niveles fallidos. La tabla 5.10 muestra el número de cambios promedio de los aprendices de cada edad en cada nivel, coloreados de verde (pocos cambios) a rojo (muchos cambios). La fila coloreada inferior es el promedio por nivel de todos los aprendices y la última fila, simplemente a título de referencia, indica el número óptimo de cambios de cada nivel entendido como el número mínimo de cambios para obtener la solución más corta posible. La columna derecha ("Nº medio cambios") marca el promedio de los cambios de todos los niveles superados, coloreada en azul oscuro a claro en proporción al número de cambios.

La tabla muestra una relación de dificultad similar a la que se observa en indicadores anteriores, con los niveles más complejos exigiendo un mayor número de cambios en el espacio de trabajo.

Edad	N i1	Nivel															N f15	Nº medio cambios
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
8	70	2,8	14,7	14,2	21,1	14,6	56,6	7,0	22,5	21,8	90,0	210,0					0	43,2
9	96	2,0	12,4	18,9	20,8	19,0	54,3	6,3	28,9	39,3	33,6	123,6	26,5	17,7			0	31,0
10	292	1,7	8,4	12,6	16,4	12,2	50,8	6,8	20,1	37,6	47,0	58,7	32,8	47,5	35,0		0	27,7
11	184	2,4	7,8	9,6	13,4	9,0	44,3	5,3	18,6	34,6	55,5	58,3	28,8	86,7	67,8	46,1	10	32,6
12	157	2,5	8,1	11,0	16,6	8,1	35,9	5,5	15,4	26,8	39,8	50,6	22,3	52,2	47,9	61,2	5	26,9
13	135	2,1	6,5	10,2	14,3	6,7	37,0	5,4	16,6	26,4	31,5	57,3	21,1	28,6	33,4	27,3	3	21,6
14	70	1,9	5,6	9,2	13,2	6,0	30,1	4,2	11,6	22,6	28,2	29,0	12,2	34,2	32,5	74,3	4	21,0
Todas	1004	2,1	8,6	12,0	16,1	10,5	44,4	5,9	18,6	32,2	44,2	56,6	24,3	55,2	48,8	52,1	22	28,8
nº óptimo camb.		1	3	2	5	4	14	1	4	4	5	3	3	5	5	8		4,5

Tabla 5.10. Número de cambios por edad y nivel superado.

Quedándonos con el indicador único por edad del número medio de cambios (figura 5.21), observamos de nuevo la dificultad que supone KodetuGen2 para las edades inferiores, que va disminuyendo en progresión hasta los 14 (K-W: p-valor<0,05 para los niveles 1 al 10, 12 y 13).

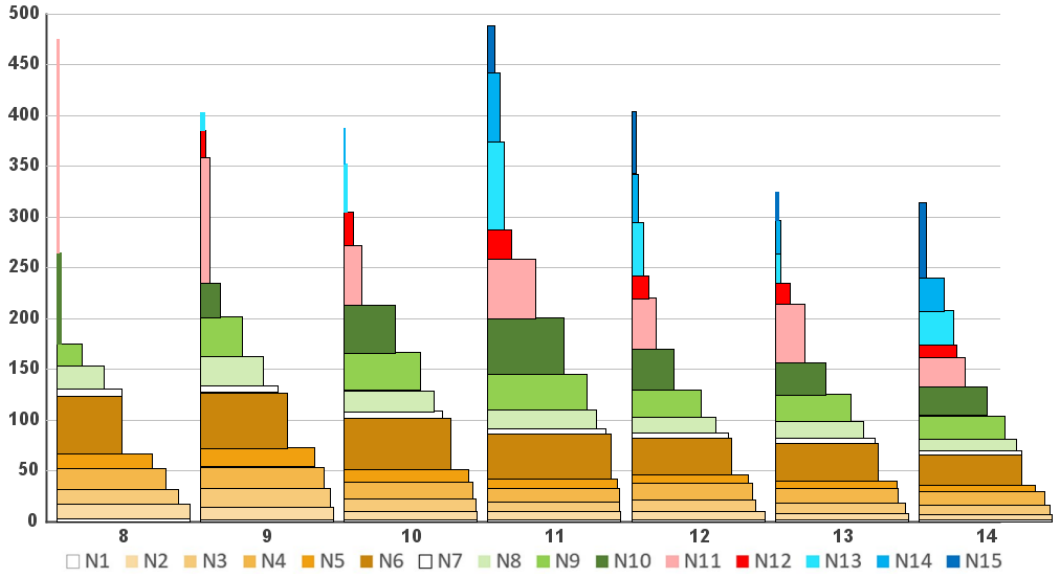


Figura 5.21. Gráfica de número de cambios promedio por edad en niveles superados.

5.3.3. Resultados en función del sexo

El 51,7% de nuestros aprendices son chicas (519) y el 48,3% son chicos (484). La distribución no es homogénea por edades, se mueve entre el 44,3% de chicas (en aprendices de 8 años) y el 66,7% de chicas (en aprendices de 12 años).

Superación de niveles

Analizamos a continuación los resultados de los aprendices de KodetuGen2 segmentados por sexo. En la tabla 5.11 podemos ver los porcentajes de aprendices que pasan cada nivel de acuerdo a su sexo (masculino "M" o femenino "F") y su progresión por niveles. Para utilizar un indicador único hemos calculado el nivel superado medio (formado por el promedio de aprendices que superan los distintos

niveles). También indicamos la diferencia porcentual de cada nivel en la fila inferior, en tonos cianes cuando es mejor para los chicos y magentas cuando es mejor para las chicas.

Sexo	N i1	Nivel															N f15	Niv.medio superado
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
F	519	100,0%	100,0%	97,1%	93,6%	89,6%	77,3%	71,9%	63,8%	52,6%	34,5%	16,6%	10,4%	6,2%	4,0%	1,7%	9	8,2
M	484	100,0%	100,0%	97,9%	94,4%	91,1%	77,7%	75,0%	67,4%	57,2%	39,5%	23,1%	12,0%	7,9%	5,0%	2,7%	13	8,5
Comparación		0,00%	0,00%	-0,82%	-0,78%	-1,52%	-0,42%	-3,13%	-3,58%	-4,63%	-4,97%	-6,57%	-1,58%	-1,69%	-0,91%	-0,95%		

Tabla 5.11. Porcentaje de aprendices que superan cada nivel, por sexo.

Se puede ver que de forma consistente los chicos se comportan algo mejor que las chicas: 8,5 frente a 8,2 de nivel superado medio, diferencia que en todos los niveles tiene la misma tendencia: se mantiene alrededor del 1% de diferencia durante todo el juego y aumenta hasta el 3-7% en los niveles intermedios 7-11. Podemos observar esta característica en la figura 5.22, con las líneas de color marcando cada uno de los colectivos (chicas y chicos) (M-W-W: W=2.456.578, p-valor < 0,001). Este dato confirma comportamiento de sexo recogido en la literatura en estas edades (Román-González, Pérez-González y Jiménez-Fernandez, 2017).

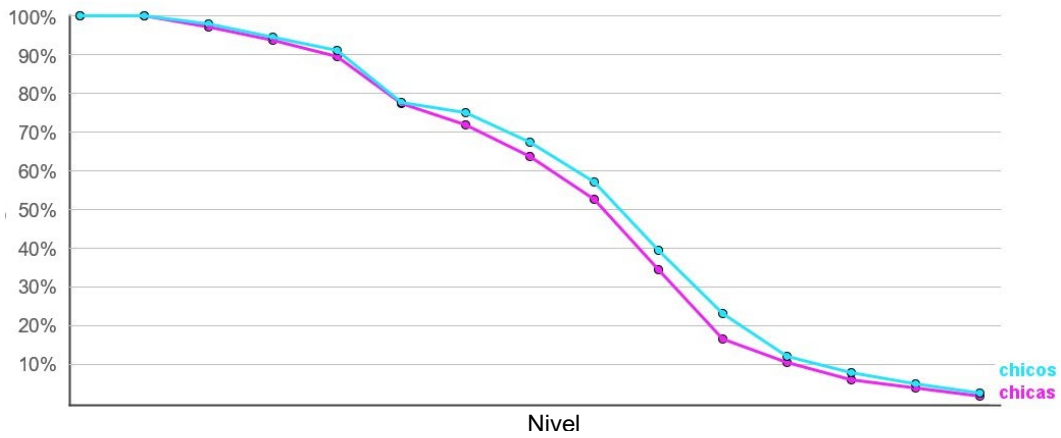


Figura 5.22. Porcentaje de aprendices que superan cada nivel por sexo (cian chicos, magenta chicas).

Contrastando estos datos por edades, la tendencia es la misma excepto en las edades mayores (13 y 14) en las que se invierte ligeramente, sin llegar a ser significativa. En cuanto al comportamiento relativo en cada nivel, observando el porcentaje de aprendices que tiene éxito en cada nivel con respecto a los que lo inician (no al total), las chicas se comportan mejor en los niveles 6, 12 y 14.

Tiempos

Veamos cómo ese mejor comportamiento en el rendimiento directo se refleja en el resto de variables. En la tabla 5.12, vemos las diferencias de los tiempos medios de solución entre chicas ("F") y chicos ("M"). Cada celda es la mediana de tiempo utilizada por chicos o chicas en ese nivel, en minutos, y la fila inferior muestra la

diferencia (cian para mejor comportamiento -menor tiempo- de chicos, magenta para mejor comportamiento de chicas).

Sexo	N i1	Nivel														N f15	Sum. T / nivel
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
F	519	1,9	0,9	1,6	1,8	1,0	4,7	0,9	1,3	2,5	3,2	5,7	2,4	3,7	4,1	1,2	9	36,8
M	484	1,8	0,7	1,2	1,3	0,7	4,4	0,8	1,1	2,2	2,6	4,6	1,4	3,1	2,0	3,8	13	31,6
Comparación		0,1	0,2	0,4	0,5	0,3	0,3	0,1	0,2	0,4	0,6	1,2	1,0	0,6	2,1	-2,6		

Tabla 5.12 Medianas de tiempos (minutos) por sexo en niveles superados, sumados en última columna.

Vemos que en general los chicos utilizan menos tiempo que las chicas (5,2 minutos de diferencia en el total) y esta tendencia se da en todos los niveles, excepto en el nivel 15 en el que las 9 chicas que acaban son sustancialmente más rápidas (2,6 minutos más rápido) que los 13 chicos (M-W-W: p-valor < 0,05 en los niveles 1 a 12 y 15). La edad media de esas chicas es 11,6, la de los chicos 12,4.

Número de ejecuciones y de cambios

Respecto al número de ejecuciones, la tendencia se mantiene. Los chicos tienen mejores resultados generales en este indicador (3,1 ejecuciones totales menos que las chicas) y solo en el nivel 15 se invierte (4,8 ejecuciones totales menos que los chicos) (M-W-W: p-valor < 0,05 hasta el nivel 9).

En el caso del número de cambios realizados en los niveles exitosos, también los chicos realizan en general menos cambios (424,6 frente a 433,5), excepto en el nivel 15 (47 cambios más los chicos) y también en el 6 (2,8 cambios más los chicos) (M-W-W: p-valor < 0,05 hasta el nivel 7).

5.3.4. Resultados en función del conocimiento previo de programación

Calculamos los mismos indicadores estadísticos ahora segmentando por el conocimiento previo de programación, declarado por los aprendices antes de empezar la prueba. El 79,6% indica no tener conocimientos previos y el 20,4% restante sí. La diferencia es muy elevada en los más pequeños (el 92,9% de los aprendices de 8 años no tiene conocimientos previos) pero se mantiene alrededor del 80% en el resto de edades.

Niveles superados

Se observa en general que los aprendices con conocimiento previo de programación declarado se comportan mejor (nivel medio al que llegan 8,9) que los que no lo tienen (8,2). Esta diferencia empieza a notarse en los primeros niveles, se invierte levemente en el nivel 6 (el trabajoso secuencial donde la experiencia previa es menos importante que la constancia de resolver paso a paso el laberinto) y a

partir de ahí va aumentando en los niveles de bloques estructurados (llegando a un 13,22% de diferencia en el nivel 11). Es lógico si consideramos que la experiencia previa a menudo dará mayor perspectiva de cómo componer estructuras repetitivas y alternativas básicas (M-W-W: W=1.454.884, p-valor < 0,001) (figura 5.23).

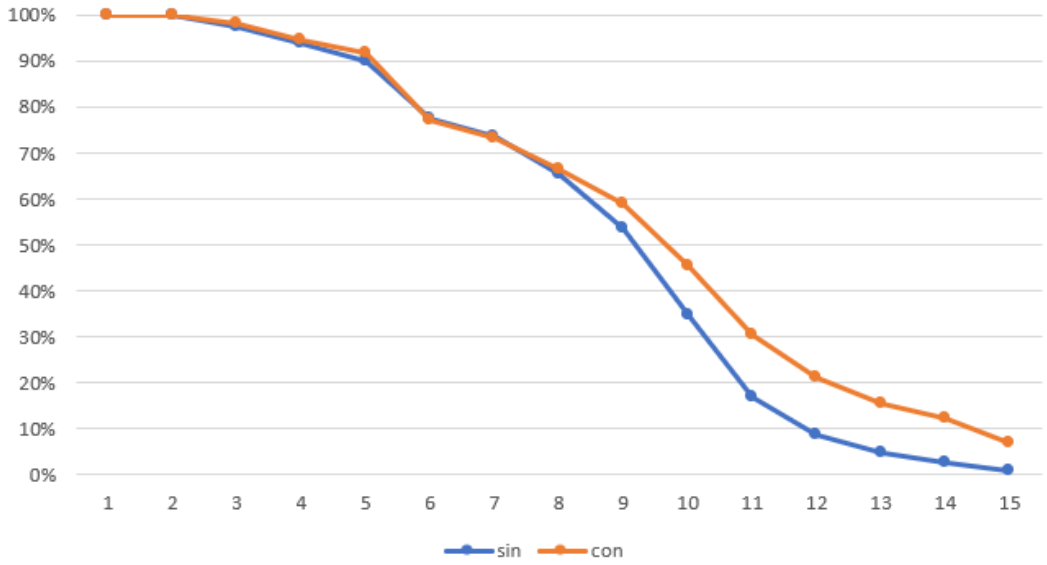


Figura 5.23. Porcentaje de superación (eje Y) en cada nivel (eje X) en función del conocimiento previo de programación.

Por edades, la diferencia es menos apreciable en las edades inferiores y mucho más notoria a partir de los 11 años (nivel medio superado: 0,3 de diferencia en menores de 11 años, 1,1 de diferencia a partir de 11 años).

Tiempos, número de ejecuciones y cambios

En el resto de indicadores se observa la misma tendencia ya comentada, con una particularidad que vamos a reseñar. En todos los casos (tiempo, número de ejecuciones y número de cambios) los aprendices con conocimiento previo de programación se comportan mejor (valores inferiores) que los que declaran no tenerlo. Y esto ocurre sistemáticamente en todos los niveles, pero se observa la tendencia no significativa de la inversión en el último nivel. Se puede observar en la figura 5.24 que la forma gráfica de la evolución en los 15 niveles sigue un patrón muy similar en los tres indicadores.

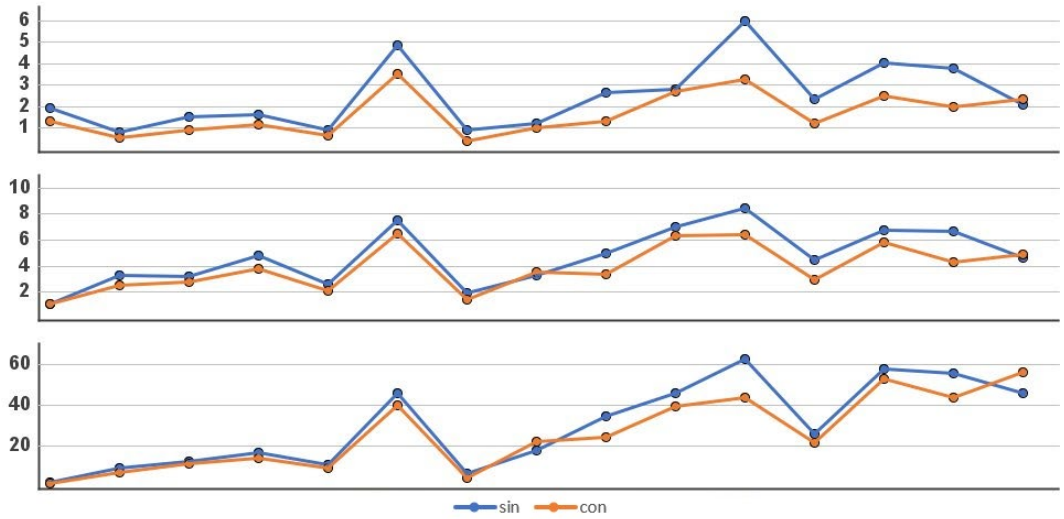


Figura 5.24. Evolución por niveles (eje X) para los aprendices sin conocimientos de programación (azul) y con ellos (naranja). Eje Y: En el gráfico superior tiempo en minutos (mediana); en el centro nº de ejecuciones (media); en el inferior nº de cambios (media).

El tiempo mediana total de diferencia a favor de los aprendices con conocimientos previos es 12,6 minutos (sustancial, ya que es un 33% más rápido que los aprendices sin conocimientos previos: 25,1 minutos frente a 37,7). La mejora en el número de ejecuciones es de 0,8 por nivel exitoso (3,9 contra 4,7, un 17,7%). En cuanto al número de cambios, mejoran en 4 cambios por nivel superado (25,9 contra 29,9, un 13,2% mejor).

Como es lógico esperar, el comportamiento general de los aprendices es mucho mejor si ya tienen conocimientos previos de programación. Sin embargo, en el nivel 15 que es un reto complejo independientemente de lo ya conocido, su rendimiento es peor (aunque llegan en mucha mayor proporción a ese nivel: un 6,8% de los aprendices con conocimientos previos llegan al nivel 15, frente a solo un 1,0% de los que no los tienen).

5.3.5. Resultados en función de la afinidad con la tecnología

Para realizar el análisis de los resultados de los aprendices desde el punto de vista de la afinidad con la tecnología revisamos los valores de la pregunta que contestan antes de la prueba ("*¿Cuánto te gusta la tecnología?*"), que está escalada del 1-mínimo al 10-máximo. Calculando el coeficiente de Spearman vemos que no hay correlaciones consistentes entre esa afinidad y los resultados de progreso, tiempo, ejecuciones o cambios.

Partiendo de esa consideración, reagrupamos la afinidad para analizarla en cuatro categorías: 0-Baja (valores 1 o 2), 1-Media-Baja (valores 3 al 5), 2-Media-Alta (valores 6 al 8) y 3-Alta (valores 9 y 10). Solo 26 aprendices se declaran en la categoría baja (2,6%), 67 en la media-baja (6,7%), 293 en la media-alta (29,2%) y los

618 restantes (61,6%) en la alta. Es decir, más del 90% de los aprendices declaran una afinidad alta o media-alta y casi la mitad (47,8%) ponen el valor máximo.

Dado que el tema trabajado es el PC y que la herramienta es tecnológica, cabe esperar un mejor desempeño de los aprendices más inclinados hacia la tecnología. Sin embargo, hay algunos resultados que no van en esta dirección.

Superación de niveles

En la propia solución del juego, el mejor grupo no es el de la categoría alta (nivel medio conseguido 8,4) sino la media-alta (nivel 8,6). Sí responden peor las media-baja y baja en el orden esperado (7,5 y 6,7) (K-W: chi-cuadrado 13,598, df = 3, p-valor = 0,0035).

En cuanto a la evolución en los 15 niveles, las dos categorías altas tienen mejor respuesta en casi todo el desarrollo del juego, pero en los tres niveles finales se iguala. Cabe destacar que, de hecho, la categoría alta es la peor en esos tres niveles y la baja la mejor en ellos, pero la muestra es tan pequeña que las diferencias no son significativas.

Tiempos, ejecuciones y cambios

La afinidad por la tecnología no parece tener influencia en el tiempo esperable de solución del juego. Tiene una influencia no significativa en el número de ejecuciones y cambios: cuanto más gusta la tecnología, más soluciones se prueban y más cambios se plantean.

5.4. Conclusiones

Tras las modificaciones realizadas en la herramienta, la segunda versión de generalización aporta resultados interesantes. Se observa cómo la incorporación del doble laberinto provoca un rendimiento global inferior de los aprendices. La generalización incluye una dimensión de complejidad adicional que hace más difícil completar los retos en el tiempo disponible: podría ser deseable disponer de más tiempo para afrontar los retos de programación y la edad objetivo ideal parece ser mayor que con el juego original, comportándose mejor los aprendices entre 11 y 14. Se observa un comportamiento no esperado en los aprendices de edad 11 y proponemos algunas razones posibles que lo explicarían.

Vemos que los indicadores de tiempo, número de ejecuciones y número de cambios mantienen una significativa correlación, a la vez que informan de algunos detalles más finos del aprendizaje en los distintos niveles.

El comportamiento de los chicos es ligeramente mejor que el de las chicas, con algunas inversiones en los niveles superiores.

Los aprendices que declaran conocimiento previo de programación se comportan sustancialmente mejor que los que no lo tienen en todos los indicadores, con una tendencia no significativa de inversión en el último nivel.

Para confirmar estas tendencias y contestar a nuestras preguntas de investigación, nos queda la parte fundamental de realizar la comparación entre el comportamiento de los aprendices en la plataforma convencional (Kodetu) y la que incorpora la generalización (KodetuGen2). En el siguiente capítulo, estableceremos esta relación y propondremos con ello nuestras conclusiones finales.

The ego lives through comparison. How you are seen by others turns into how you see yourself.

Eckhart Tolle

Capítulo

6

Comparativa de la generalización

Realizado el experimento presentado en el capítulo anterior, podemos comparar los resultados de los aprendices en las tres plataformas utilizadas, especialmente para conocer cuáles son las diferencias entre el juego convencional Kodetu y el juego con generalización mejorado KodetuGen2.

Profundizaremos en este capítulo en esa comparativa y presentaremos tras ello la discusión sobre las preguntas de investigación ya planteadas.

6.1. Preparación

Definimos en el capítulo anterior el concepto de tipo Kodetu, de acuerdo a la clasificación en la que diferenciamos la complejidad de los niveles con 4 categorías A, B, C, D. En cualquiera de las tres versiones de Kodetu, se pasa secuencialmente por una serie de niveles en este orden: primero los niveles del tipo A (7 en Kodetu y KodetuGen, 6 en KodetuGen2), luego los del tipo B (2 en Kodetu y KodetuGen, 3 en KodetuGen2), después 2 niveles de tipo C y finalmente 3 de tipo D (tabla 6.1).

Dejamos de considerar los dos niveles 1, tutorial de inicio y 8 (7 en KodetuGen2), tutorial de bloques repetitivos, que no están diseñados para validar la capacidad de los aprendices sino para enseñarles los puntos significativos del juego y su operativa.

	tut	A						*tut*	B			C		D		
		A1	A2	A3	A4	A5	A6		B1	B2	B3	C1	C2	D1	D2	D3
K	1	2	3	4	5	6	7	8	9	10		11	12	13	14	15
KG	1	2	3	4	5	6	7	8	9	10		11	12	13	14	15
KG2	1	2	3	4	5		6	7	8	9	10	11	12	13	14	15

Tabla 6.1. Codificación por tipo Kodetu de los niveles de las tres versiones.

De esta forma, en lugar de obtener valores estadísticos por nivel, lo haremos por tipo Kodetu. El comportamiento de cada aprendiz en todos los niveles incluidos en cada uno de los 4 tipos se recogerá en un indicador único por cada tipo, teniendo de esta manera 4 mediciones por aprendiz en lugar de 15. Para realizar comparaciones estas mediciones son más oportunas, ya que abordan niveles que tienen complejidad teórica similar, aunque siempre con el elemento de la generalización presente en el caso de KodetuGen2 y KodetuGen, con las connotaciones ya comentadas para KodetuGen en el cuarto capítulo (muchos aprendices no están en realidad realizando un proceso mental de generalización).

De acuerdo a los análisis realizados con los datos separados de cada una de las plataformas, ya hemos visto cuáles son los elementos que influyen en el desempeño de los aprendices. En base a ello, definimos los siguientes indicadores para cada uno de los 4 tipos:

- Coeficiente de niveles superados. Como el número de niveles no es homogéneo, lo normalizamos en el intervalo [0,1]. Cuando un aprendiz haya intentado el primer nivel de un tipo pero no lo haya superado, se indicará con un 0. Si pasa algunos de los niveles incluidos, el valor sería proporcional al número de niveles del tipo: por ejemplo, en Kodetu hay 6 niveles incluidos en el tipo A. Si se superan 2, se obtendrá un 0,33. Si se superan todos los niveles de un tipo, el valor será 1.
- Número medio de ejecuciones en nivel resuelto. Considerando todos los niveles que se superen dentro de cada tipo, se calcula por cada aprendiz su media de número de ejecuciones (intentos de solución). Obviamente, este valor al menos es 1 ya que es imprescindible pulsar alguna vez el botón para superar el nivel. En cada caso, se calcula como el número de todas las ejecuciones que ese aprendiz ha pulsado en los niveles superados del tipo, dividido por el número de niveles superados de ese tipo.
- Tiempo total medio en nivel resuelto. Del mismo modo, calculamos el tiempo total invertido por cada aprendiz en todos los niveles superados de un tipo y lo dividimos por el número de niveles superados. La precisión medida es de milisegundos, aunque lo mostraremos en minutos para facilitar su interpretación. En paralelo, calculamos el indicador inverso de número de niveles resueltos por minuto que a veces resulta un poco más claro para entender o explicar alguno de los resultados.
- Número medio de ejecuciones por minuto en niveles resueltos. Nos planteamos cómo de importante es la frecuencia con la que los aprendices intentan ejecutar sus códigos y si hay diferencia dependiendo del experimento. Para ello, calculamos también el cociente entre el número de ejecuciones en todos los niveles resueltos de cada tipo y el tiempo invertido en esos niveles (en minutos).

- Número medio de cambios de código extra. El número de cambios de código ha sido tenido en cuenta en el análisis previo, pero ahora se da la circunstancia que en las tres versiones de Kodetu cada nivel tiene diferentes códigos que resuelven el laberinto. Por este motivo el número de cambios no puede compararse directamente. Calculamos entonces cuál es el número de cambios mínimos en cada nivel: la diferencia de longitud entre el código inicial y el código con la solución final (suponiendo que no se hace ningún borrado o modificación). Así, podremos calcular la diferencia entre los cambios realmente efectuados y esos cambios mínimos, de manera que cada aprendiz tendrá un valor de cambios "extra" que es cero en el mejor de los casos y mayor cuantos más cambios se hayan efectuado. Se calcula también la media de los cambios extra por nivel (cociente de los cambios extra realizados en los niveles resueltos de cada tipo entre el número de niveles resueltos).
- Relación entre longitud de código y la longitud menor de código correcto. Habiendo analizado los códigos previamente, sabemos que la menor longitud de código no es necesariamente la mejor manera de resolver cada nivel. Sin embargo, sí queremos medir si hay diferencias apreciables dependiendo de la generalización. Para ello calculamos este indicador como cociente entre la suma de longitudes de código suministrados como solución de los niveles de cada tipo y la suma de las longitudes de los códigos más cortos que solucionan esos niveles. Este valor será entonces 1 o superior.

Todos estos valores pueden contener valores vacíos ("blanco") en el caso en el que el aprendiz no haya llegado a iniciar ningún nivel de ese tipo (por ejemplo, si no se ha acabado el nivel 12 -último del tipo C-, se tendrá un valor blanco en el tipo D en todos sus indicadores).

De cara al análisis, se han calculado también una serie de indicadores que finalmente no han sido utilizados en nuestra investigación por no resultar de interés. Los nombramos para informar de que están disponibles en las tablas de análisis realizadas:

- Número medio de ejecuciones por nivel no resuelto.
- Tiempo total medio por nivel no resuelto.
- Número medio de ejecuciones por minuto en nivel no resuelto.
- Número de cambios en el nivel no resuelto.
- Relación de pasos por nivel y pasos óptimos por nivel en las soluciones finales correctas (tanto en el primero como en el segundo laberinto).
- Número medio de bloques de código muerto (no ejecutados) en las soluciones finales correctas (en ambos laberintos).

Datos intrasujeto e intersujetos

A partir de este momento, tenemos una serie de datos *intrasujeto*, calculados para cada aprendiz como un promedio de su comportamiento en una serie de niveles (los incluidos en uno de los cuatro tipos). Esta "puntuación" nos permitirá evaluar el comportamiento de cada aprendiz en cada nivel. Hemos decidido que este dato sea una **media ponderada** porque parece más oportuno considerar todos los niveles de cada tipo en su conjunto (total de tiempo entre número de niveles o total de ejecuciones entre tiempo total, por ejemplo) y estos valores nos darán los indicadores para medir las tendencias de cada aprendiz ("tendencia a realizar una ejecución", "eficiencia de cambios de código", etc.).

Es importante notar que cuando hagamos el análisis *intersujetos* del colectivo, no hay razones para dar más peso a unos aprendices que a otros por lo que calcularemos "tendencia promedio a realizar una ejecución" o "eficiencia promedio de cambios de código" como una **media aritmética no ponderada**, lo que nos dará una visión de la muestra de aprendices conjunta, diferenciada por versiones de Kodetu y por el resto de los factores considerados.

6.2. Resultados

6.2.1. Comparación entre niveles

Antes de realizar la comparación de resultados con los tipos A-D definidos en las tres plataformas, observamos algunos aspectos generales para poder abordarla con la mayor precisión posible.

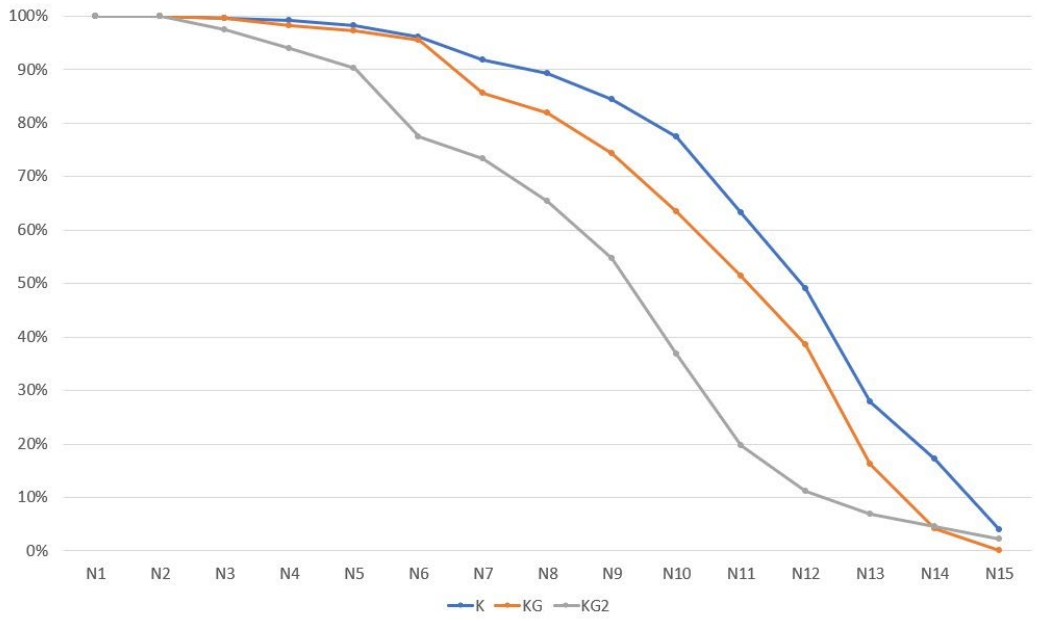


Figura 6.1. Evolución de los aprendices en los 15 niveles por cada versión de Kodetu.

En primer lugar, desde el planteamiento del diseño de Kodetu con generalización suponíamos que resolver los distintos niveles del juego sería más costoso para la mayor parte de los aprendices. Analizando los datos, vemos que eso efectivamente es así y es muy significativo (figura 6.1). Desde el nivel 3 (el primero con generalización), se constata la diferencia de aprendices que van superando el juego (un 2,2% más en Kodetu que en KodetuGen2), que va creciendo hasta los niveles complejos de estructuras combinadas (nivel 12: 37,8% de aprendices más en Kodetu que en KodetuGen2), a partir de ahí la diferencia disminuye pero se mantiene muy significativa (en el nivel final son casi el doble, 4,0% frente a 2,2%).

Comparando los datos de los aprendices con éxito también encontramos diferencias. Calculamos los tiempos medios de los aprendices que han conseguido superar cada nivel (figura 6.2). La relación nivel a nivel, como ya hemos comentado, no tiene equivalencia uno a uno en el caso de KodetuGen2. Pero sí podemos observar la evolución y los tiempos acumulados para reconocer un valor objetivo de comparación de dificultad. Un juego medio "perfecto" de cada una de las tres plataformas llevaría 44,7 minutos en Kodetu, un minuto más en KodetuGen y más de 11 minutos adicionales en KodetuGen2.

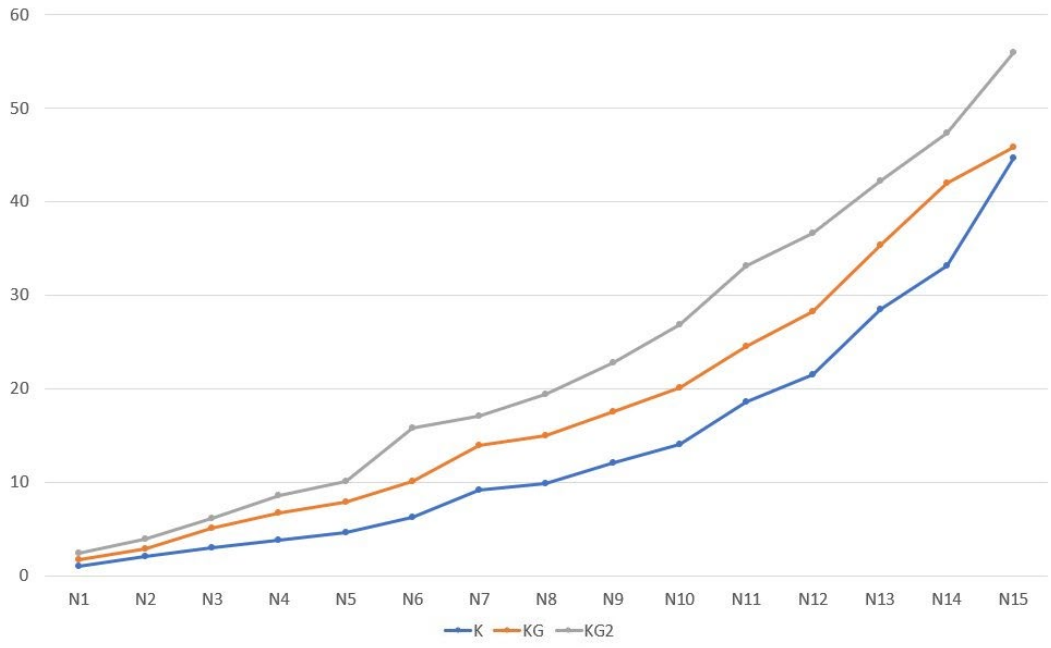


Figura 6.2. Tiempo acumulado medio (mins.) de solución de los niveles de las tres herramientas en las secuencias de niveles 1 a 15.

A lo largo de las comparaciones de este capítulo, esta dificultad relativa de KodetuGen2 se irá observando en los distintos indicadores.

Dado que la comparación nivel a nivel no es siempre equiparable, a lo largo de esta sección establecemos el paralelismo en los niveles equivalentes de acuerdo a la organización descrita (A=secuencias, B=repetitivas, C=repetitivas y alternativas

simples, D=repetitivas y alternativas dobles) y numerando los niveles en secuencia, podemos compararlos como se observa en la tabla 6.2.

	A1	A2	A3	A4	A5	A6	B1	B2	B3	C1	C2	D1	D2	D3	Total
K	1,0	0,9	0,8	0,8	1,7	2,8	2,1	2,0		4,5	3,0	6,9	4,6	11,6	42,9
KG	1,2	2,2	1,5	1,2	2,2	3,9	2,6	2,5		4,4	3,8	7,0	6,6	3,8	43,1
KG2	1,5	2,2	2,4	1,5		5,7	2,3	3,3	4,2	6,2	3,5	5,6	5,2	8,6	52,1

Tabla 6.2. Tiempo medio (mins.) de los niveles resueltos por reto, comparados por nivel más similar.

Observamos estos mismos datos en la figura 6.3, donde no hay puntos para los niveles sin equivalencia (el nivel 6 de Kodetu y KodetuGen en KodetuGen2 y el tercer nivel de repetitivas de KodetuGen2 que no tienen Kodetu ni KodetuGen). En esta figura, donde eliminamos también los niveles tutoriales (el 1 y el primero con repetitivas), es posible ver que la dificultad, medida en tiempo de solución, no es homogénea en las tres plataformas. En general, los niveles de KodetuGen2 se resuelven empleando más tiempo, exceptuando los últimos. Esto es algo que analizaremos con profundidad más adelante. También se observa un comportamiento anómalo de los aprendices en el último nivel de KodetuGen, que en general es un poco más costoso que Kodetu. Esto se debe al reducido tamaño de la muestra de aprendices que llega a resolver ese nivel (N=4), con lo que ese punto temporal (solo 3,8 minutos de media en resolver un nivel que se tarda 11,6 en Kodetu) se debe principalmente a la brillantez de los escasos aprendices que lo consiguen.

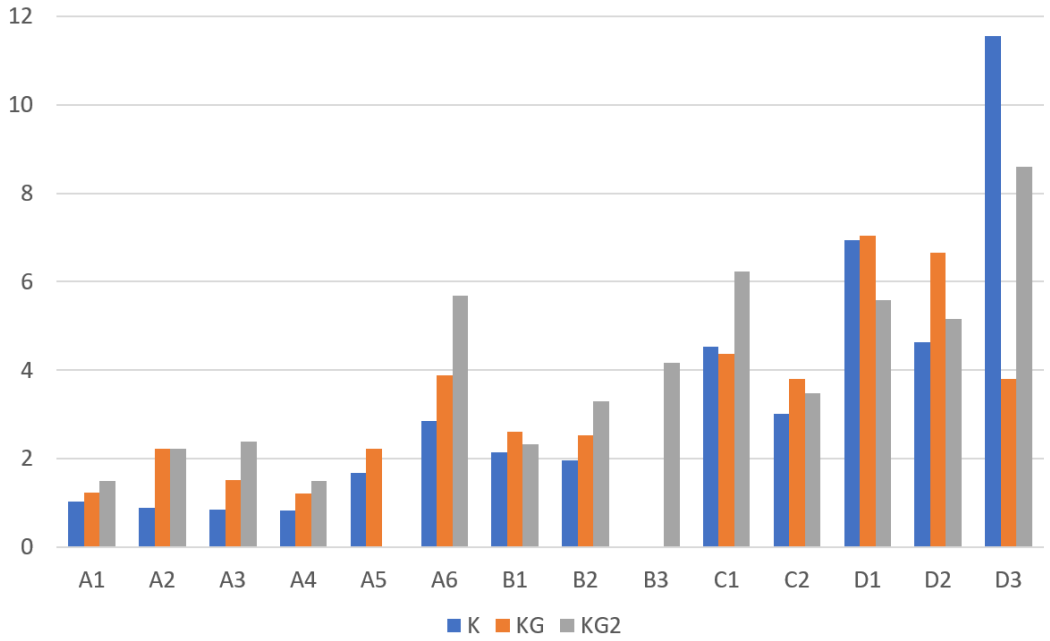


Figura 6.3. Tiempo medio (mins.) de solución de niveles de las 3 plataformas comparadas nivel a nivel.

6.2.2. Resultados en función del tiempo de juego

Las consideraciones que ya hicimos sobre el tiempo de juego se aplican de forma similar en todas las versiones de Kodetu. Nos preguntamos si hay diferencia en los tiempos de juego en las tres versiones y vemos que las distribuciones son bastante similares. Especialmente entre Kodetu y KodetuGen2, que es la comparación que más nos va a interesar, la diferencia tanto en la media como en la mediana es de aproximadamente un minuto y la desviación típica es muy similar (tabla 6.3).

	M	Me	D.T.
K	28,790	28,286	12,077
KG1	29,223	28,760	10,217
KG2	29,770	29,443	12,186

Tabla 6.3. Estadísticos descriptivos de los tiempos de juego (minutos) en las tres versiones.

Como era de esperar, ante el mismo planteamiento de sesiones controladas los aprendices tienden a utilizar un poco más de tiempo en KodetuGen2 (ese minuto adicional), dada su complejidad.

Aunque el rendimiento es peor, observemos sin embargo en la figura 6.4 cómo se distribuye el nivel medio que se supera dependiendo del tiempo que se juegue (en intervalos de 5 minutos). Se observa en Kodetu una tendencia a estabilizarse a partir de los 45 minutos (no se mejora el rendimiento por jugar más tiempo), mientras que en KodetuGen2 la tendencia se mantiene creciente: haría falta más tiempo para enfrentarse con similares garantías de éxito en KodetuGen2. La diferencia de rendimiento entre ambas se va reduciendo a partir de los 50 minutos.

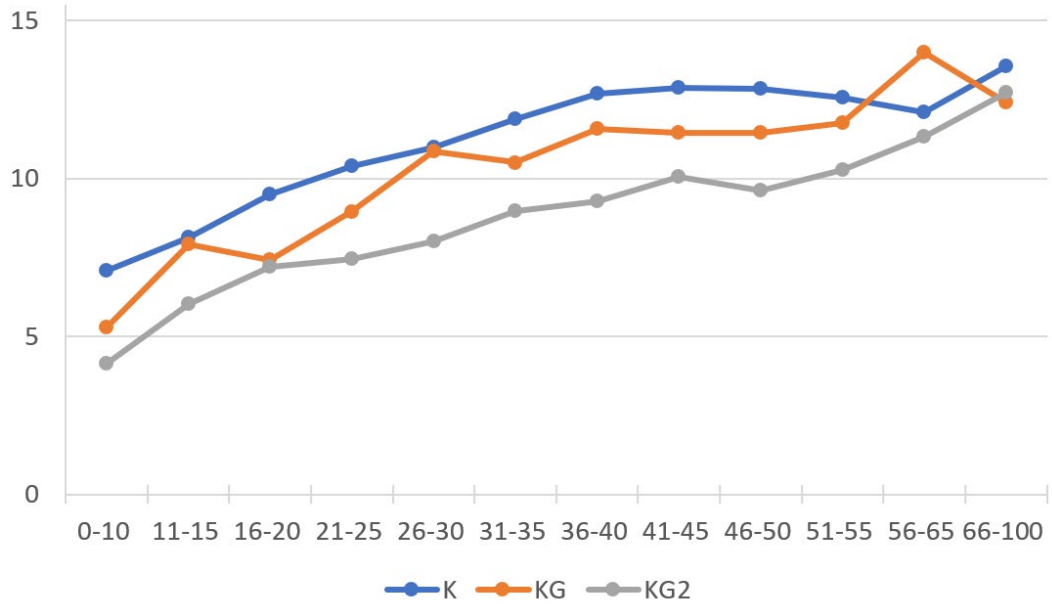


Figura 6.4: Nivel medio superado (eje X) en función de los intervalos de tiempo de juego (eje Y).

6.2.3. Resultados globales

Analizamos primero el comportamiento de los aprendices en general en las tres versiones de Kodetu, con las distintas variables calculadas. Posteriormente haremos algunas anotaciones segmentándolos.

Para ello, tendremos en cuenta a los aprendices que sobreviven en cada uno de los tipos (tabla 6.4), donde ya observamos la dificultad de KodetuGen2 con respecto a las otras dos versiones.

	A	B	C	D
K	100,0%	88,4%	75,9%	46,3%
KG	100,0%	81,1%	62,9%	37,6%
KG2	100,0%	72,6%	36,0%	10,8%

Tabla 6.4. Supervivencia de aprendices por tipo y versión de Kodetu.

De acuerdo a nuestras hipótesis de investigación, queremos valorar si la generalización es un aspecto de influencia significativa en el desarrollo del PC. Por eso diseñamos KodetuGen2 con los mismos niveles finales (13 a 15, tipo D). De esta forma, intentaremos valorar especialmente cómo el aprendizaje que ocurre resolviendo los niveles previos A-C afecta a la solución de esos niveles de tipo D.

Coeficiente de niveles superados

El primer elemento importante es el rendimiento directo de los aprendices en niveles superados. Para ello, observamos el indicador de superación de niveles que hemos calculado en el rango 0-1 por cada uno de los tipos de niveles (tabla 6.5).

	Global		A	B	C	D
K	0,70		0,97	0,92	0,74	0,35
KG	0,63		0,96	0,85	0,72	0,18
KG2	0,51		0,92	0,72	0,43	0,42
Total	0,65		0,96	0,87	0,70	0,34

Tabla 6.5. Comportamiento de los aprendices en cada tipo y versión de Kodetu, en el intervalo [0-1] (0 = Ningún nivel de ese tipo superado, 1 = Todos los niveles de ese tipo superados).

Para facilitar la interpretación utilizaremos una gráfica similar a la ya empleada en el análisis de KodetuGen2, pero ahora solo con los 4 tipos (en lugar de los 15 niveles). Para ello usamos un color para cada uno de los tipos, mantenemos el concepto de columnas (una columna por cada segmento diferenciado: Kodetu, KodetuGen, KodetuGen2, de izquierda a derecha) y las cajas decrecientes (indicando el ancho el porcentaje de aprendices supervivientes en cada tipo y el alto la medición en escala de la variable, en este caso niveles superados [0-1]) (figura 6.5).

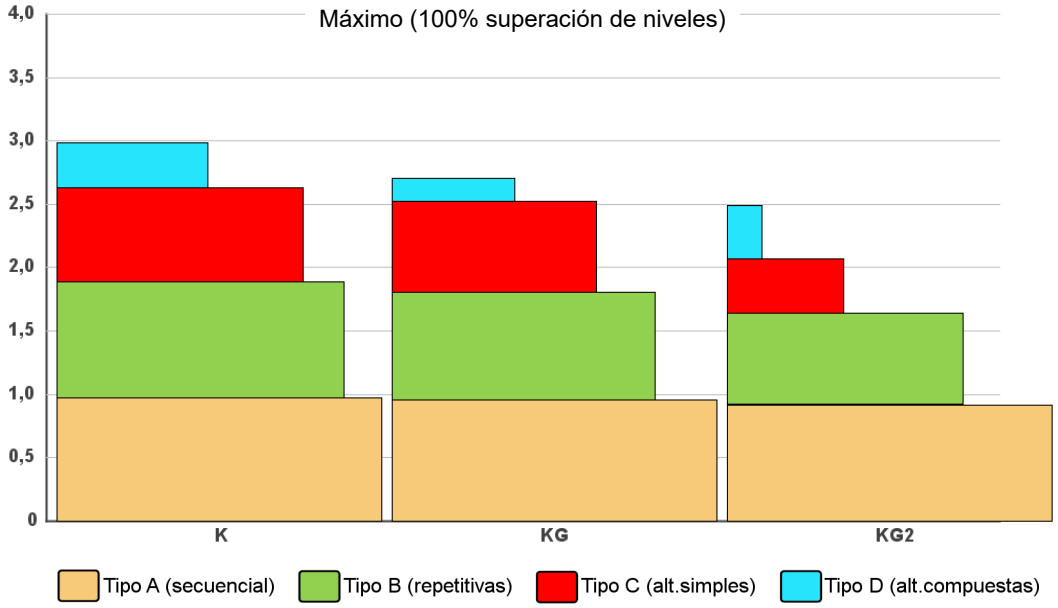


Figura 6.5. Comportamiento medio de los aprendices en la superación de niveles por tipos, en las tres versiones de Kodetu.

Vemos en la anchura de las barras cómo en KodetuGen2 los aprendices abandonan mucho antes el juego (por ejemplo en el tipo C solo llega un 36% en KodetuGen2 -caja roja derecha- frente al 75,9% de Kodetu -caja roja izquierda-). Sin embargo, hay un elemento clave que desarrollaremos en las siguientes páginas, el comportamiento en el tipo D.

Tanto en Kodetu como en KodetuGen, el rendimiento de los aprendices decrece según se avanza en el juego (y cae especialmente en el tipo D). Sin embargo, en KodetuGen2 el rendimiento se mantiene en el tipo D, tanto que **se pone por encima del rendimiento correspondiente de las otras dos versiones** (0,45 frente a 0,32 de Kodetu. K-W: p-valor < 0,001) (figura 6.6). **Entrenarse jugando con niveles que incluyen generalización hace más difícil el proceso, pero prepara mejor para enfrentarse a los problemas más abstractos de programación**.

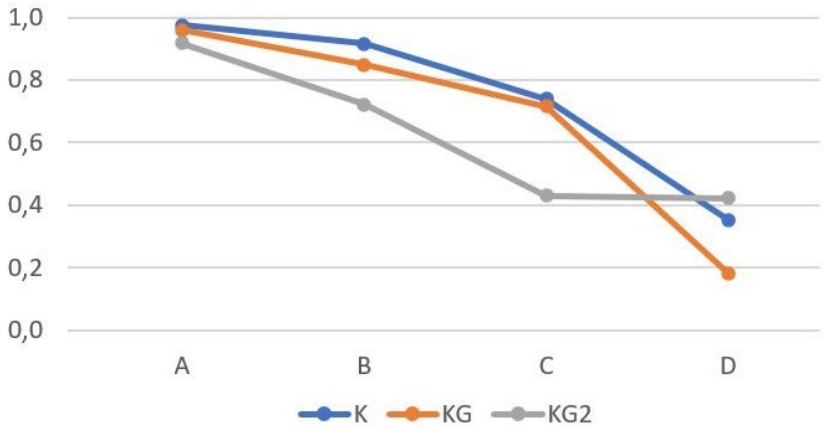


Figura 6.6. Comportamiento medio de los aprendices en la superación de niveles por tipos (A-D), en las tres versiones de Kodetu.

Tiempo en nivel superado

Dentro de cada tipo de niveles, hemos calculado el tiempo medio que cada aprendiz invierte en los niveles solucionados (tabla 6.6). Como es esperable, solucionar niveles en KodetuGen2 lleva más tiempo (2,98 minutos frente a 1,99 de Kodetu y 2,39 de KodetuGen).

	Global		A	B	C	D
K	1,99		1,35	2,06	3,80	6,27
KG	2,39		1,97	2,55	4,04	5,96
KG2	2,98		2,61	3,16	5,71	5,30
Total	2,25		1,69	2,31	3,98	6,17

Tabla 6.6. Tiempos (mins.) de solución de nivel, global y de acuerdo a su tipo, diferenciado en las 3 versiones de Kodetu y el general ("Total").

No obstante, el aprendizaje que ocurre en los niveles A-C permite que los aprendices supervivientes de tipo D en KodetuGen2 no solo bajen el tiempo que les llevaban los niveles de tipo C (5,3 minutos por 5,71), sino que pasan a ser los aprendices de las tres versiones de Kodetu que **menos tiempo necesitan para solucionar los niveles de tipo D** (5,3 minutos frente a 5,96 en KodetuGen y 6,27 en Kodetu. K-W: p-valor $\leq$ 0,0175). En la figura 6.7 se observa cómo la única bajada de tiempo en la evolución secuencial es la que experimenta KodetuGen2 entre los niveles C y D.

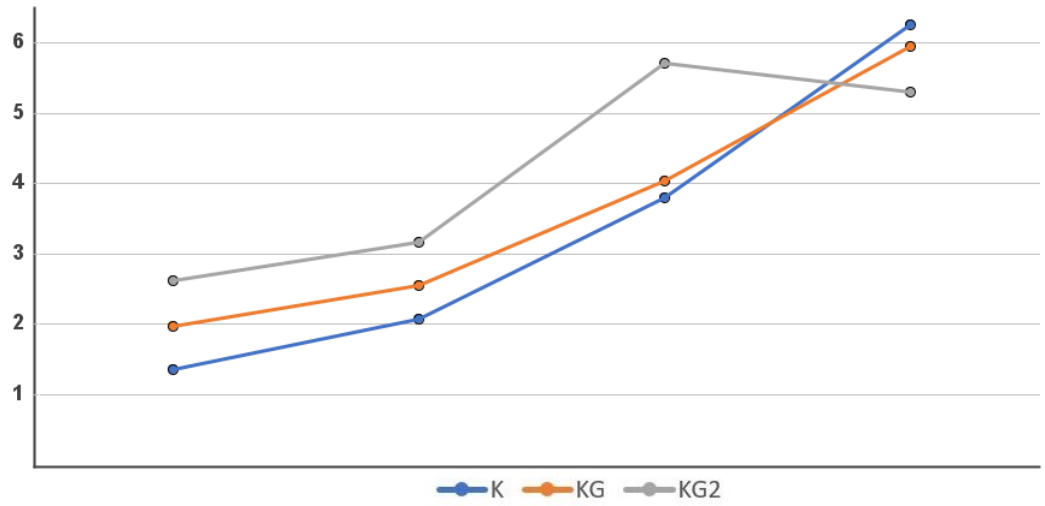


Figura 6.7. Tiempos medios (minutos) en nivel superado por tipo A-D, por versión de Kodetu.

Ejecuciones en nivel superado

Veamos cuántas ejecuciones necesitan los aprendices de las diferentes versiones. En la tabla 6.7 podemos observar el número medio de ejecuciones que usan para resolver un nivel dentro de cada tipo (recordemos que 1 es el mínimo en una solución perfecta en la que no hace falta probar soluciones fallidas antes de dar con la correcta).

	Global	A	B	C	D
K	3,25	2,73	3,33	4,58	6,43
KG	4,31	3,88	4,59	5,84	8,35
KG2	4,42	4,07	4,58	6,59	5,99
Total	3,63	3,16	3,70	4,90	6,58

Tabla 6.7. Número de ejecuciones en nivel resuelto por tipo en las 3 versiones de Kodetu y en general.

Podemos observar (y apreciar visualmente en la figura 6.8) que de nuevo se mantiene la tendencia vista con los tiempos. Los aprendices de KodetuGen2 que llegan a los niveles de tipo D no solo hacen menos ejecuciones en esos niveles que en los de tipo C (5,99 frente a 6,59), sino que **realizan menos ejecuciones que cualquiera de los aprendices de las otras versiones** (6,43 en Kodetu y 8,35 en KodetuGen. K-W: p-valor < 0,001).

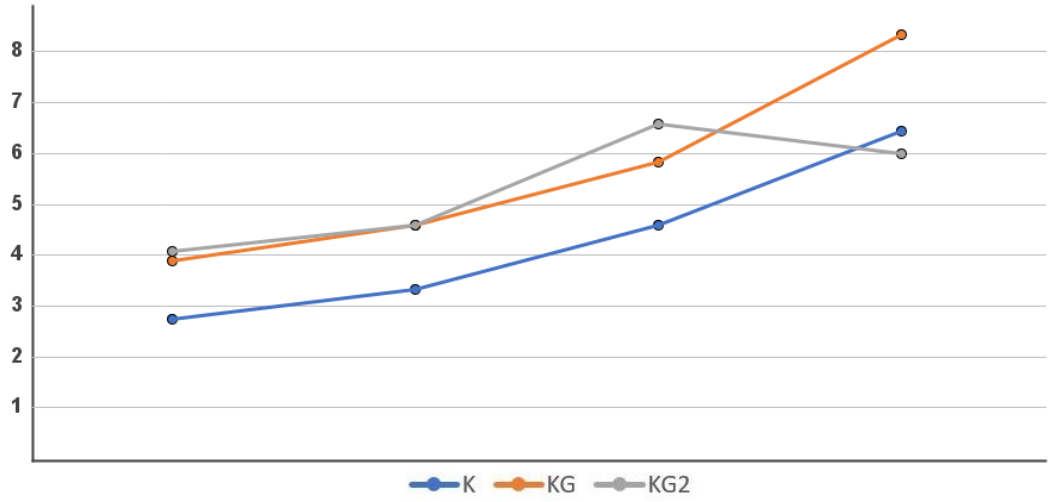


Figura 6.8. Número medio de ejecuciones en nivel resuelto por tipo, en las 3 versiones de Kodetu.

Ejecuciones por minuto en niveles superados

Consideramos también un indicador que no tiene que ver estrictamente con la eficiencia del PC, sino con la capacidad de probar que tienen los aprendices (sean pruebas correctas o incorrectas). Para ello, hemos calculado el número de ejecuciones por minuto que realizan en los niveles finalmente solucionados de cada tipo (tabla 6.8).

	Global	A	B	C	D
K	1,80	2,36	2,13	1,45	1,09
KG	1,90	2,16	2,07	1,54	1,44
KG2	1,61	1,76	1,60	1,37	1,17
Total	1,77	2,21	2,03	1,46	1,12

Tabla 6.8. Ejecuciones por minuto en nivel resuelto y tipo en las 3 versiones de Kodetu y en general.

En los tipos iniciales, los aprendices de KodetuGen2 son capaces de probar con menos frecuencia que el resto de las versiones, en los niveles de tipo D se iguala con Kodetu (1,09 en Kodetu, 1,17 en KodetuGen2) mientras la mayor frecuencia de prueba pasa a ser para KodetuGen.

Cambios extra en niveles superados

Calculamos también (Tabla 6.9) los cambios que se realizan en cada nivel superado y particularmente la diferencia con el número óptimo de cambios (de modo que cero sería el nivel perfectamente resuelto en el que solo se han puesto los bloques que se necesitan para la solución, sin ningún cambio adicional).

	Global		A	B	C	D
K	12,83		6,09	17,30	31,06	54,71
KG	13,03		8,07	20,22	31,98	56,66
KG2	16,06		11,71	23,84	44,10	48,73
Total	13,54		7,53	18,79	32,21	54,49

Tabla 6.9. Cambios extra en nivel resuelto por tipo en las 3 versiones de Kodetu y en general.

Vemos que el comportamiento en este indicador es muy similar al de tiempo en nivel resuelto: Kodetu y KodetuGen tienen un índice muy parejo, creciente de forma consistente y casi lineal desde el tipo A al tipo D. KodetuGen2, aunque es sustancialmente mayor en el desarrollo (14 cambios mayor en el tipo C), decrece en el tipo D y pasa a ser significativamente inferior al de las otras dos versiones de Kodetu. (48,73 cambios extra frente a 54,71 y 56,66). Esta es una tendencia no significativa (K-W: p-valor = 0,2702) (figura 6.9).

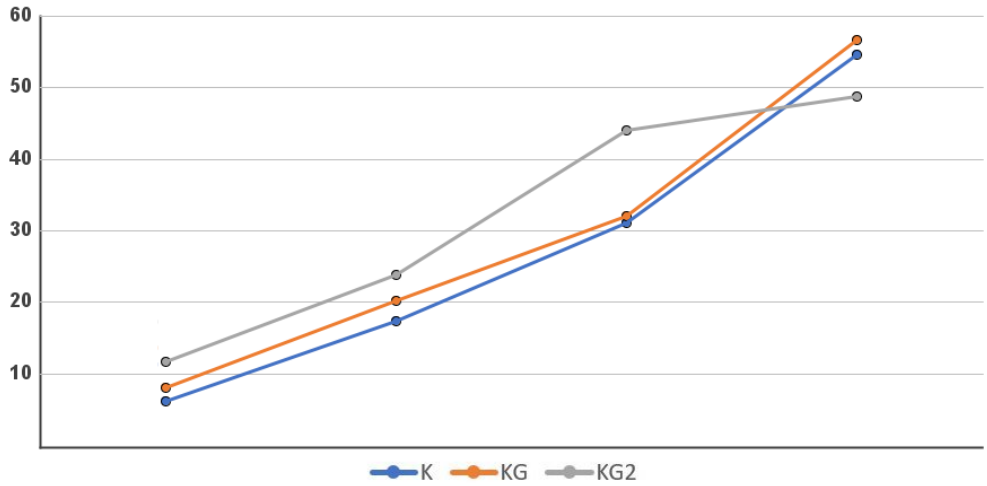


Figura 6.9. Cambios extra por tipo en las 3 versiones.

Ratio de longitud de código en niveles superados

Por último, comparamos el indicador de longitud de código, calculado como una ratio entre la longitud de código (número de bloques) final que se aporta para la solución de un nivel y el código mínimo encontrado que lo soluciona. Este indicador tiene diferencias muy pequeñas debido a las limitaciones de bloques que proporciona la propia plataforma. Por otro lado, a veces la expresividad de código necesaria para resolver un laberinto propone códigos más largos pero más generales, algo que podría incluso notarse en las versiones con generalización. En cualquier caso, 1 marcaría el código más corto y podemos observar los valores encontrados en las tres versiones en la tabla 6.10.

	Global		A	B	C	D
K	1,06		1,05	1,00	1,09	1,35
KG	1,11		1,12	1,01	1,10	1,29
KG2	1,09		1,07	1,12	1,23	1,30
Total	1,07		1,07	1,02	1,10	1,34

Tabla 6.10. Ratio de longitud de código en nivel por tipo en las 3 versiones de Kodetu y en general.

Aunque las diferencias son del orden de centésimas, vemos de nuevo el efecto de que el ratio de KodetuGen2 es bastante mayor en los tipos iniciales y pasa a ser menor en el tipo D (1,30 frente a 1,35 de Kodetu y 1,29 de KodetuGen), en este caso sin decrecer (K-W: p-valor < 0,001) (figura 6.10).

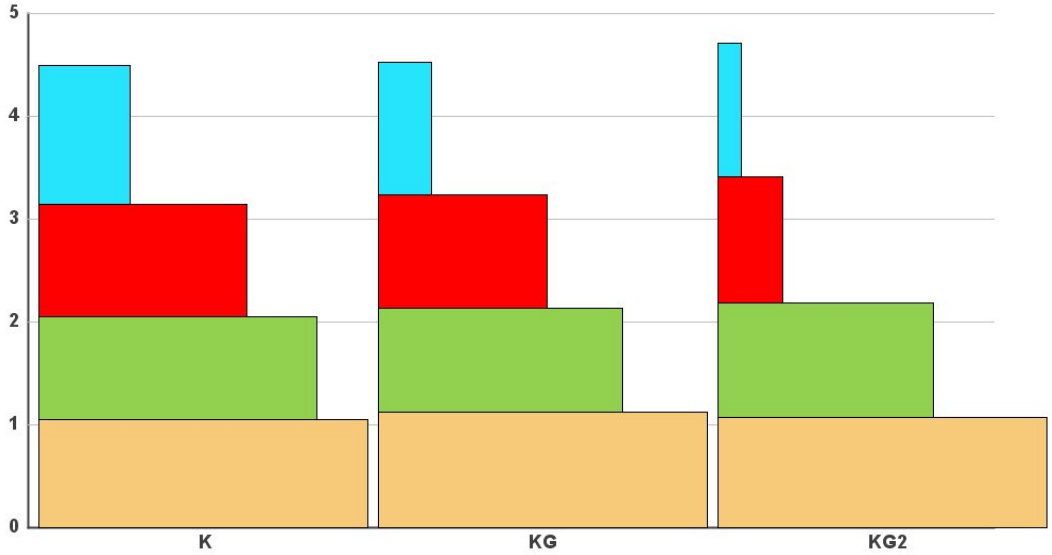


Figura 6.10. Representación en columnas decrecientes de ratio de longitud de código en nivel por tipo en las 3 versiones.

6.2.4. Sesgo de mejores aprendices

Ante los resultados comparativos y el mejor desempeño de los aprendices de KodetuGen2 en los niveles de tipo D, es evidente que esta plataforma tiene una exigencia de competencia mayor, que provoca un filtro de aprendices muy temprano (ya hemos visto, por ejemplo, que solo un 10,8% de los aprendices de KodetuGen2 llega a los niveles de tipo D, contra un 46,3% de Kodetu). Es esperable entonces que estos aprendices de KodetuGen2 se comporten mejor que los de Kodetu, ya que la propia actividad ha seleccionado a un colectivo mucho más competente.

Nos planteamos completar los análisis realizando un filtro de la muestra, para ver en qué medida KodetuGen2 resiste la comparación entre los colectivos más homogéneos posibles. Aunque las poblaciones de ambos experimentos son similares, no tenemos manera de evaluar su capacidad previa o el resto de habilidades que influyen en las primeras experiencias del PC. Sin embargo, ya realizados los experimentos sí que podemos filtrar a las submuestras que mejor desempeño hayan tenido y compararlas entre sí. Para ello, tendremos que segmentar a los aprendices de acuerdo a su comportamiento en el juego a partir de una puntuación que nos permita clasificarlos.

Análisis de componentes principales (PCA)

Para elaborar nuestra heurística de puntuación, realizamos primero un análisis de componentes principales (ACP, en inglés PCA). Es una técnica estadística que nos permite describir un conjunto de datos, partiendo de una serie de variables con correlaciones, en términos de nuevas variables (componentes) no correlacionadas. Estos componentes se ordenan por la cantidad de varianza original que describen, por lo que el análisis nos puede dar un buen indicador para reducir todas las variables que teníamos originalmente a una sola de puntuación.

Aplicando el análisis PCA sobre todas las variables que hemos venido utilizando, se produce el gráfico mostrado en la figura 6.111. En él se observan con claridad las correlaciones inversas que existen entre cada variable de superación de niveles (*TypeAOk*, *TypeBOk*...) y las variables secundarias: cambios extra de bloques (*TypeAExtraChangesAvg,* etc.), número de ejecuciones (*TypeAPlaysSolved,* etc.), tiempo de niveles resueltos (*TypeAT.Tot.Solved,* etc.) y ratio de longitud de código (*TypeACodeLengthRatio*, etc.). Hemos coloreado en la figura cada variable por su tipo Kodetu y se aprecia que el tipo A (naranja) es en el que menos correlacionan las variables. El resto de tipos (tipo B verde, tipo C rojo, tipo D azul) forman prácticamente rectas en una sola dirección.

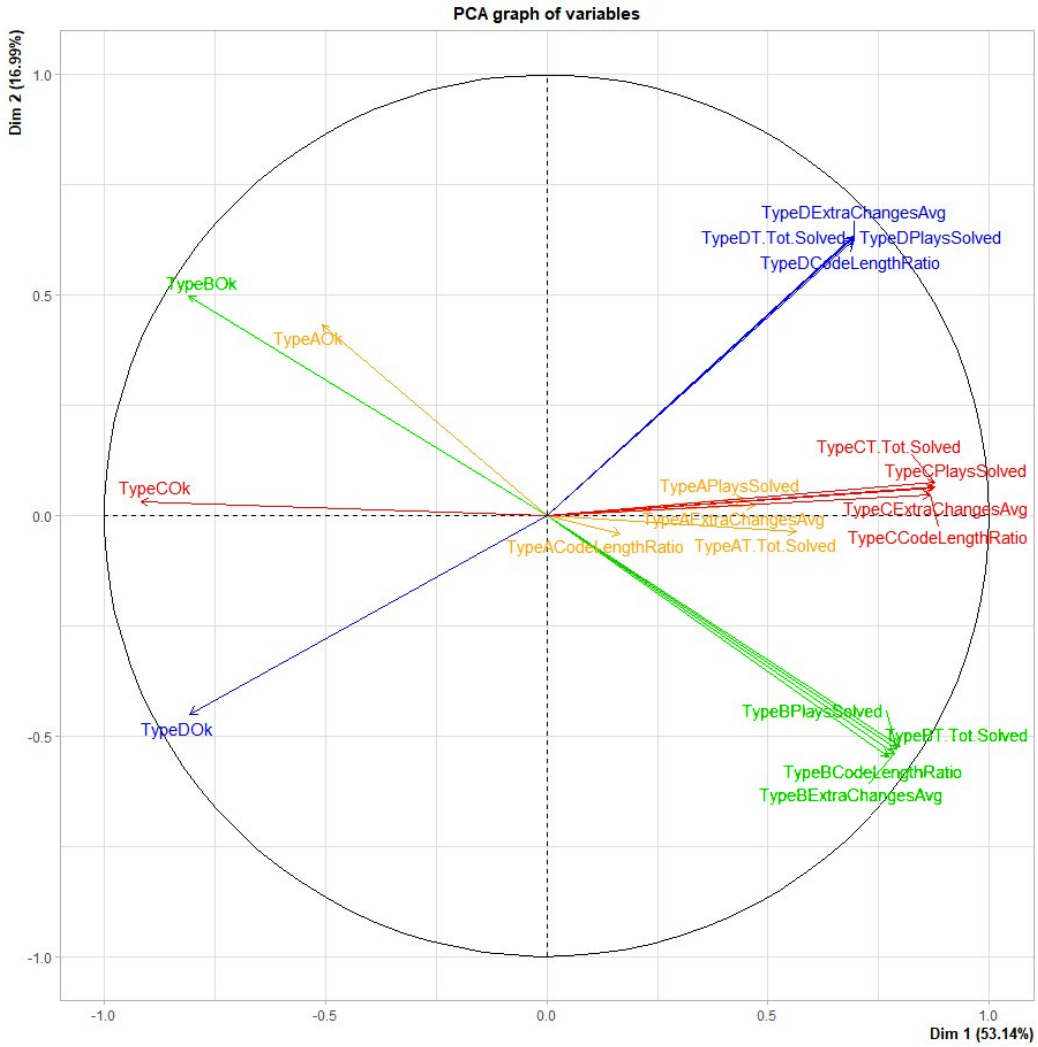


Figura 6.11. Análisis PCA de todas las variables primarias y secundarias consideradas en el análisis, coloreadas por tipo.

En el análisis hemos venido observando una variable fundamental, el índice de paso de niveles, que crece en la misma dirección del desempeño (mejor cuanto mayor valor). Las otras cuatro variables tienen la relación inversa (mejor cuanto menor valor): tiempo utilizado, ejecuciones, cambios extra, ratio de longitud de código. Realmente estas variables secundarias permiten hacer un análisis más sutil de la manera en la que programa cada aprendiz, aunque la tendencia en el colectivo es una relación inversa con el desempeño. En la figura 6.12 vemos por separado el análisis PCA para todas las variables secundarias por cada tipo, donde se observan mejor sus diferencias. En los niveles de tipo A es donde más diferencia ocurre entre las variables, en particular en el ratio de longitud de código con respecto al resto (algo lógico al ser estos niveles aquellos en los que no hay límite para el código, por tanto hay muchas variantes de código -giros inversos, caminos más largos- que permiten solucionar el laberinto con muchos más bloques). El resto de tipos manifiestan una fuerte correlación entre todas las variables secundarias.

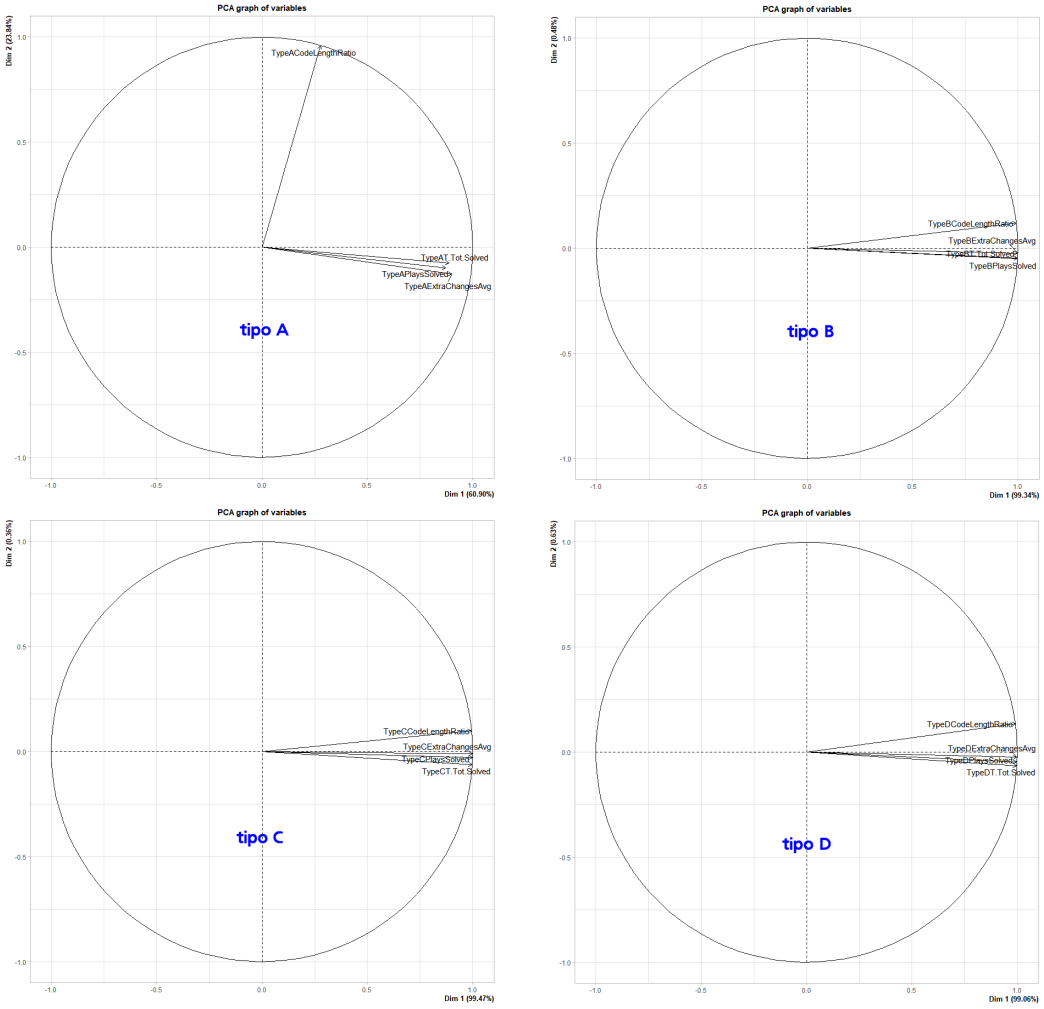


Figura 6.12. Análisis PCA de las variables secundarias en los 4 tipos (A-D).

Esta correlación no está siempre avalada por la literatura, aunque ciertamente es un análisis en el no hay todavía demasiados estudios en nuestro campo, y a menudo se busca una relación de predicción individual y no colectiva. Blikstein y cols. (2014) reseñan que "sorprendentemente, el número de cambios de código no está asociado con el desempeño en el curso".

Cálculo de puntuación

Partimos del análisis PCA global y de las ponderaciones que aporta cada una de las variables para realizar una heurística que incorpora cada uno de sus valores con el coeficiente calculado desde el PCA. Hay algunas heurísticas que calculan desempeño en la literatura (Blikstein y cols., 2014) pero no hemos encontrado ninguna propuesta que considere las variables que estamos manejando en nuestro estudio, por lo que optamos por utilizar la aproximación basada en este PCA. En nuestro caso, añadimos un peso de 16 para cada una de las variables principales, ya que entendemos que el principal componente de clasificación es el índice de

niveles superados y mantenemos un peso 1 para las variables secundarias. Para incorporar además estas, al ser inversamente proporcionales al desempeño (cuanto mayor es su valor, menos valorable), las restamos del valor máximo para cada colectivo. La fórmula de puntuación resulta de esta forma:

$$16 \cdot \sum_{t \in [A,B,C,D]} k_t \cdot ok_t + \sum_{t \in [A,B,C,D]} Ktiempo_t \cdot (Ltiempo_t - tiempo_t) + \sum_{t \in [A,B,C,D]} Kejec_t \cdot (Lejec_t - ejec_t) + \sum_{t \in [A,B,C,D]} Kcamb_t \cdot (Lcamb_t - camb_t) + \sum_{t \in [A,B,C,D]} Kratio_t \cdot (Lratio_t - ratio_t)$$

(siendo K*i* el coeficiente correspondiente y L*i* el límite de valor correspondiente)

Tras calcular la "puntuación" de cada aprendiz con esta nueva fórmula, podemos ahora reordenar a todos los aprendices de acuerdo a esta puntuación, aunque para ello diferenciamos cada una de las versiones de Kodetu: no se trata de obtener una clasificación de quiénes son los mejores aprendices, sino poder determinar quiénes están incluidos en un porcentaje concreto de los mejores. Dicho de otro modo: esta puntuación no nos clasifica entre sí los sujetos de los tres experimentos, pero sí nos clasifica con bastante precisión los aprendices dentro de cada experimento.

Ahora podemos filtrar el colectivo al punto deseado. Hacemos varias pruebas reduciendo el grupo de aprendices tomando un porcentaje de los primeros clasificados según la puntuación. Observamos que reduciendo hasta el 30% de los mejores, hay desviaciones leves pero la tendencia se mantiene como se ha observado: los aprendices de KodetuGen2 se comportan mejor en los niveles del tipo D, en todas las variables.

Buscamos un umbral máximo para la comparación y, dado que estamos comparando el comportamiento en los niveles de tipo D, calculamos el porcentaje de aprendices de KodetuGen2 que empiezan los niveles de tipo D: el 11,2%. Seleccionamos ese mismo porcentaje de los mejores aprendices de cada uno de los tres tipos de Kodetu. En este colectivo (figura 6.13) los aprendices de KodetuGen2 tienen peor desempeño de solución de niveles de tipo D que los de Kodetu (0,42 frente a 0,79) y resuelven menos niveles de tipo D por minuto (0,11 contra 0,16).

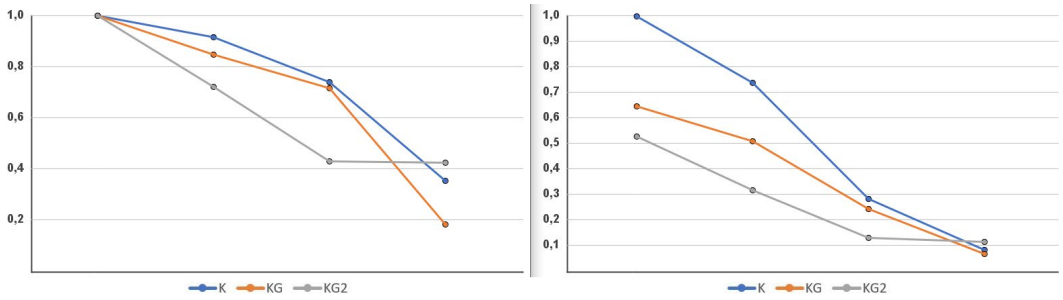


Figura 6.13. Progreso en niveles (izquierda) y niveles superados por minuto (derecha) considerando el 11,2% de los mejores aprendices en cada versión.

En las otras variables observadas (figura 6.14) también se modifica la tendencia. Los mejores aprendices de KodetuGen2 tardan más tiempo en nivel D (0,77 minutos más), hacen más ejecuciones (1,6) y más cambios extra (15,52). Sin embargo, siguen siendo mejores que los de Kodetu en el ratio de longitud de código (1,297 frente a 1,352).

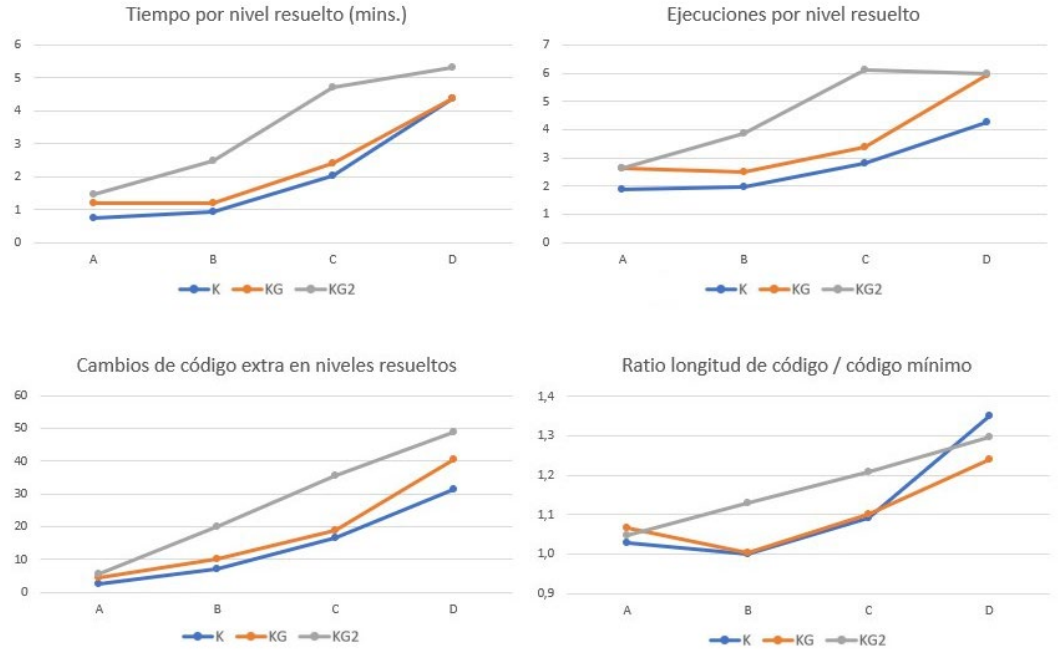


Figura 6.14. Evolución de tiempos, ejecuciones, cambios de código y ratio de long. de código (de izda. a dcha. y de arriba abajo) considerando el 11,2% de los mejores aprendices en cada versión.

Nuestra interpretación es que KodetuGen2 entrena mejor a sus aprendices pero obviamente tiene una dificultad mayor debida a sus dos laberintos y a la generalización que requieren, que hace que aprendices de similar capacidad obtengan peores resultados. A pesar de ello, aun considerando esta dificultad adicional y aprendices de similar perfil, hay algunos indicadores en los que mejora el desempeño. Por ejemplo, considerando nivel a nivel, **los mejores aprendices en KodetuGen2 son significativamente mejores en tiempo y en número de ejecuciones resolviendo el difícil nivel 15**, precisamente el que más necesita la habilidad de generalización que estamos analizando (figura 6.15).

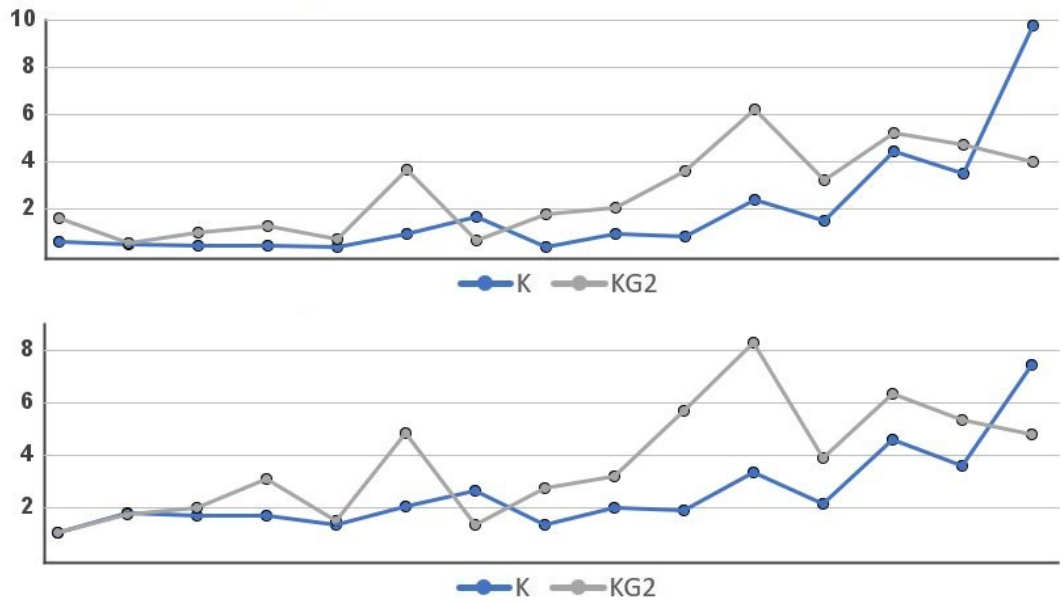


Figura 6.15. Tiempo en nivel resuelto (arriba) y ejecuciones en nivel resuelto (abajo) considerando el 11,2% de los mejores aprendices en Kodetu y KodetuGen2.

6.2.5. Resultados en función de la edad

Completamos el análisis de los resultados comparativos con las distintas segmentaciones utilizadas. Indicamos solo los resultados significativos y observamos en primer lugar la edad.

La diferencia más relevante con respecto a la edad tiene que ver con el comportamiento de los aprendices de 11 años ya reseñado en el análisis de KodetuGen2. Este comportamiento no se da en Kodetu, donde la progresión es bastante lineal (figura 6.16) aunque tiende a detenerse en los 13-14 años. Una posible explicación a estos datos es que 13-14 sea la mayor edad objetivo donde Kodetu es una apropiada primera experiencia del PC. Como comentamos, KodetuGen2 no parece ser un reto aceptable como primera experiencia del PC en edades inferiores a 11 años.

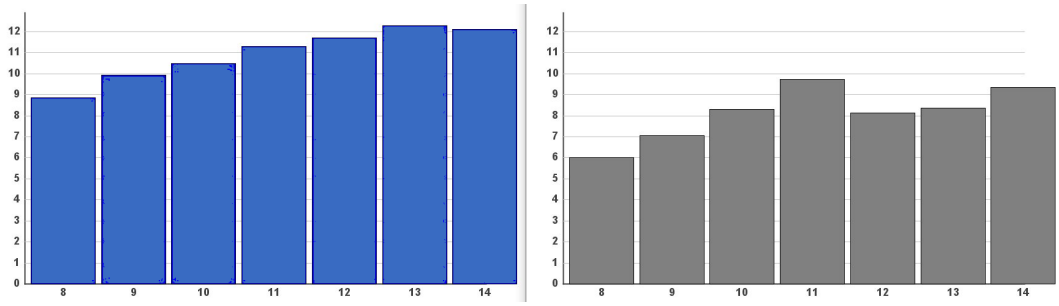


Figura 6.16. Evolución de nivel medio resuelto con respecto a la edad en K (izda.) y KG2 (dcha.).

La dificultad excesiva que conlleva para las edades bajas (8-10) KodetuGen2 se observa en varias de nuestras variables. En particular, en el número extra de cambios realizados en niveles resueltos, se ve como la linealidad decreciente de

Kodetu para todas las edades contrasta con el pico de esfuerzo que les significa a los pocos aprendices de 8-9 años de KodetuGen2 que llegan a los niveles de tipo C (figura 6.17).

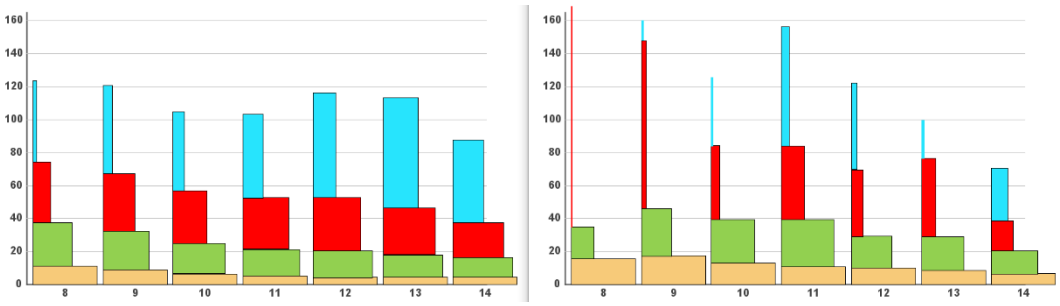


Figura 6.17. Cambios extra en niveles resueltos por edad. K (izda.) vs. KG2 (dcha.).

La manera en la que las distintas edades se enfrentan a los retos es diferente cuando observamos con el número de ejecuciones en nivel resuelto (figura 6.18). En Kodetu, los aprendices de 12-14 años obtienen más rendimiento probando más veces mientras que los 11 parece ser una edad óptima desde el punto de vista de la prueba (el menor número de ejecuciones). En KodetuGen2, los aprendices de 12-14 años son más reflexivos y se enfrentan a la abstracción de la generalización probando significativamente menos que edades más tempranas.

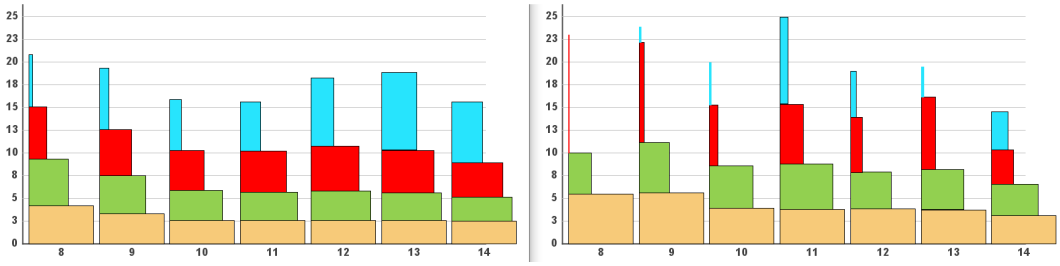


Figura 6.18. Número de ejecuciones en niveles resueltos por edad. K (izda.) vs. KG2 (dcha.).

6.2.6. Resultados en función del sexo

En general, los chicos resuelven mejor que las chicas las distintas versiones de Kodetu. Sin embargo, en el original la diferencia es más apreciable (nivel medio de las chicas 10,59 contra 11,31 de los chicos, un 6,8% peor) que en KodetuGen2 (8,19 vs. 8,51, un 3,9% peor). Curiosamente, en KodetuGen la diferencia es aún mayor (8,4%, 9,65 vs. 10,46. K-W: p-valor < 0,001). Este dato va en la dirección observada en Kodetu de que las chicas se comportan mejor si hay situaciones de mayor abstracción (generalización) y la planteada erróneamente en KodetuGen no responde a ello.

Por tipos, en la figura 6.19 vemos como las diferencias son siempre favorables a los chicos. Son pequeñas al inicio, se acrecientan en los niveles B y C y vuelven a disminuir en los niveles D, tanto en Kodetu como en KodetuGen2. Relevante observar que dentro de esos niveles D, también en ambos casos, las chicas mejoran a los chicos significativamente en el comportamiento en el nivel 15.

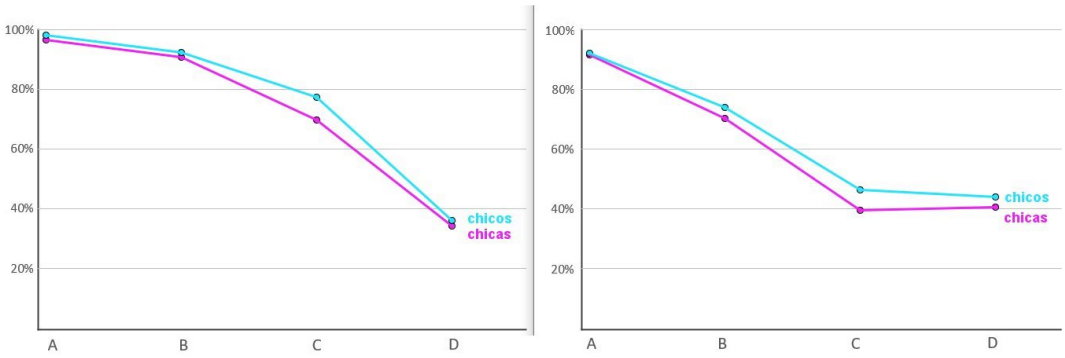


Figura 6.19. Superación de niveles en función del sexo (K izda., KG2 dcha.).

En todos los casos parece observarse que los chicos hacen más cambios y las chicas son más reflexivas. Vemos que los cambios por minuto (figura 6.20) de las chicas son siempre unos pocos menos que los de los chicos. Esta diferencia se acrecienta en los momentos de trabajo mecánico (tipo A de Kodetu, diferencia de 1,0 cambios por minuto) y en los de mayor carga abstracta (tipo D de KodetuGen2, 1,2 cambios por minuto).

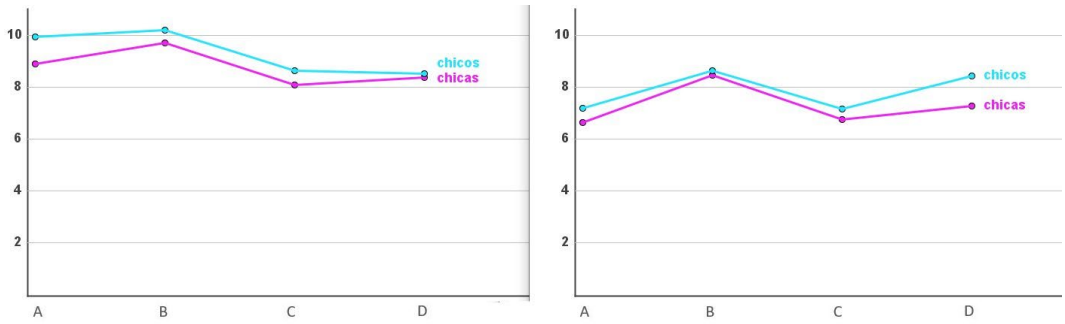


Figura 6.20. Cambios por minuto (eje Y) en función del sexo (K izda., KG2 dcha.).

Los chicos son más rápidos en todos los casos en tiempo medio de solución de nivel. La diferencia media en Kodetu es de 8,4 segundos, mientras que en KodetuGen2 es tres veces mayor, 24,3 segundos. Los niveles con mayor diferencia son los más abstractos, C de KodetuGen2 (33,2 segundos) y D (35,2 segundos) (figura 6.21).

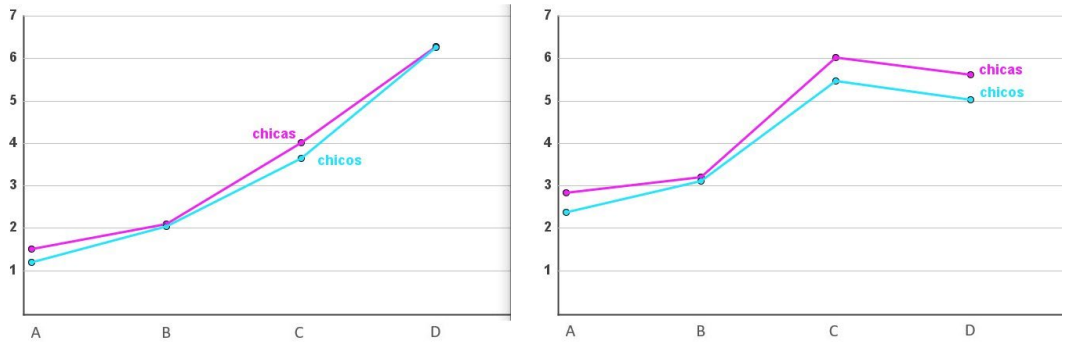


Figura 6.21. Tiempo (eje Y, minutos) en nivel resuelto en función del sexo (K izda., KG2 dcha.).

Otro punto relevante es el número de ejecuciones en nivel, que es bastante homogéneo entre chicas y chicos, excepto en los niveles D de KodetuGen2, donde hay una importante diferencia de ejecuciones, 6,4 frente a 5,3. Parece que los chicos aprendices de KodetuGen2 mejoran especialmente al pasar de los niveles C a los D (figura 6.22). La misma tendencia se da en los cambios extra.

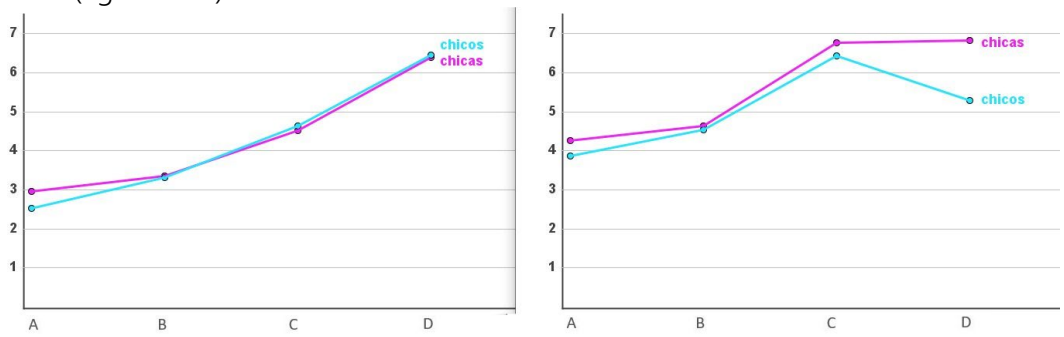


Figura 6.22. Número de ejecuciones (eje Y) en nivel resuelto en función del sexo (K izda., KG2 dcha.).

6.2.7. Resultados en función del conocimiento de programación

El conocimiento previo de programación tiene mucha más influencia en KodetuGen2 que en Kodetu. Los niveles superados son mejores para los aprendices con conocimiento de código por centésimas en Kodetu y por décimas en KodetuGen2, especialmente en los niveles C y D (figura 6.23).

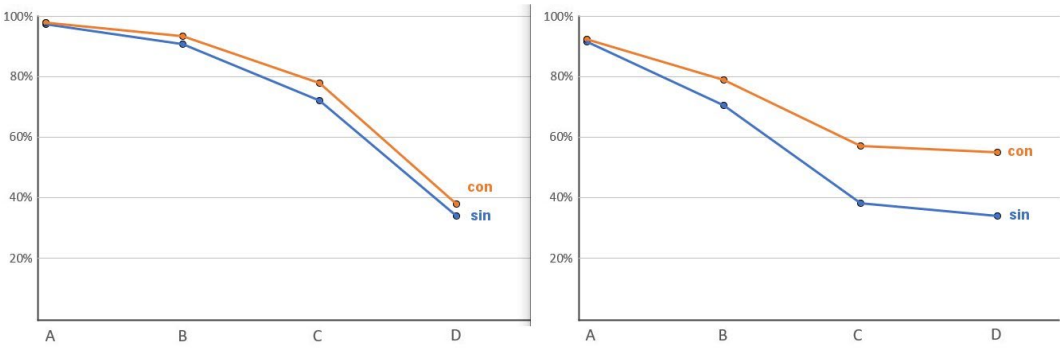


Figura 6.23. Coeficiente de solución de niveles por conocimiento de código (K izda., KG2 dcha.).

Aunque en ambos casos realizan más cambios extra los aprendices sin conocimientos de código, es reseñable cómo en KodetuGen2 esta diferencia es mucho mayor en los niveles C y en menor medida en los D (figura 6.24).

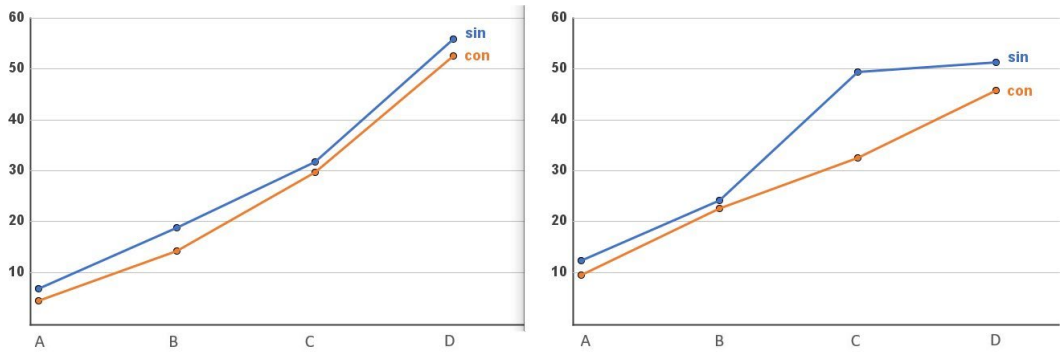


Figura 6.24. Cambios extra diferenciados por conocimiento de código (K izda., KG2 dcha.).

Este mismo efecto se percibe en el número de ejecuciones y en el tiempo invertido en solucionar los niveles, aunque en este caso la diferencia de tiempo es igual de reseñable en los niveles C y D (figura 6.25).

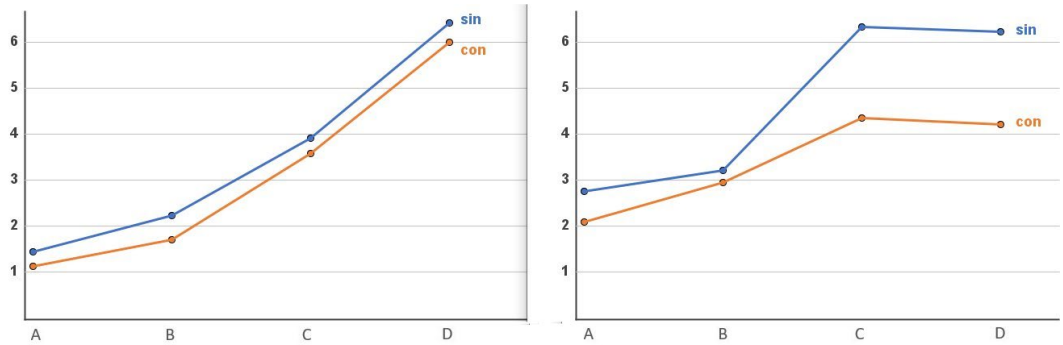


Figura 6.25. Tiempos de solución por conocimiento de código (K izda., KG2 dcha.).

6.2.8. Resultados en función de la afinidad con la tecnología

Dividimos de nuevo la escala de afinidad con la tecnología en cuatro rangos en lugar de 10 (1-2, 3-5, 6-8 y 9-10) y observamos que tanto en Kodetu como en KodetuGen2 las distribuciones no son uniformes (menos del 10% de los aprendices declaran afinidad baja o media-baja), lo que hace que la significancia estadística de esta comparación sea mínima.

Aún así, es bastante contraintuitivo que el desempeño de los aprendices en los niveles más complejos, sea prácticamente inverso al esperable (peor cuanta más afinidad con la tecnología), tanto en Kodetu como en KodetuGen2 (con más significancia en los niveles abstractos de KodetuGen2, donde este efecto es palpable (figura 6.26). En el cómputo general este efecto se minimiza porque la mayoría de los aprendices en los niveles A y B sí tienen el comportamiento esperable. Parece que la afinidad a la tecnología no es un factor relacionado con la capacidad de abstracción del PC.

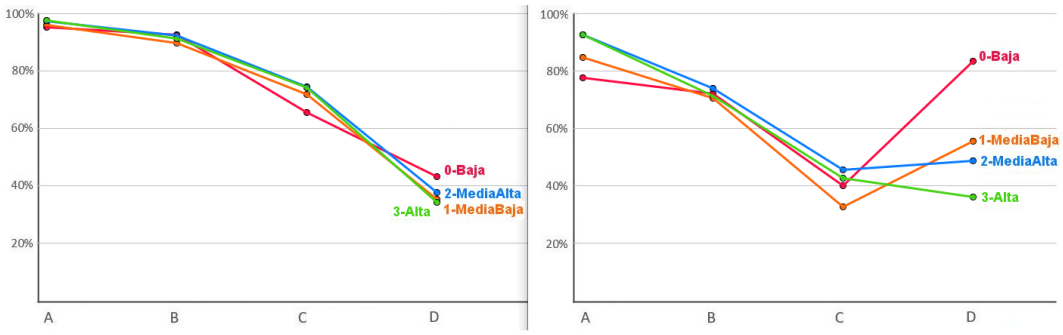


Figura 6.26. Niveles superados por afinidad tecnológica (K izda., KG2 dcha.).

Este efecto de inversión se observa también en el resto de los parámetros analizados, donde afecta mucho más a KodetuGen2. Por ejemplo, en el caso del número de ejecuciones en nivel resuelto, donde los aprendices de los niveles C y D con mayor afinidad por la tecnología necesitan muchas más ejecuciones para superar el reto (Figura 6.27).

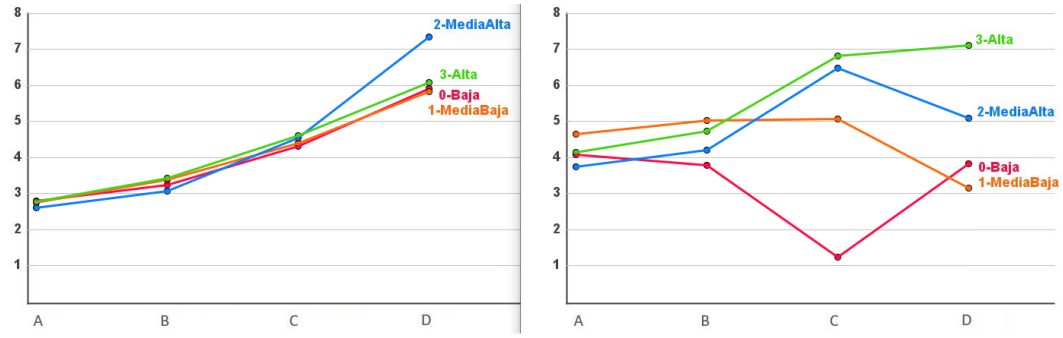


Figura 6.27. Ejecuciones en nivel resuelto según afinidad con la tecnología (K izda., KG2 dcha.).

6.2.9. Comparación de código exitoso

Al igual que en Kodetu, nos interesan también las particularidades de los códigos que los aprendices usan para resolver los retos de los distintos niveles. Realizamos comparaciones entre los códigos con los que los aprendices resuelven KodetuGen2 y los que utilizan los aprendices de Kodetu, para valorar si hay diferencias significativas.

En primer lugar, hay dos niveles particulares en los cuales se han introducido consideraciones específicas en el diseño que revisamos. En el nivel 11 (con correspondencia con el nivel 11 de Kodetu), se ampliaron los límites de bloques a 8 en lugar de 5. Nos preguntábamos en qué medida influyen estos límites en la expresividad de los aprendices: cuántos códigos diferentes válidos son capaces de generar. La respuesta es muy significativa: en KodetuGen2 se obtienen 33 códigos diferentes de 198 éxitos (el 16,7% si medimos la expresividad en porcentaje, donde el 100% significaría una absoluta expresividad: todo códigos diferentes). En Kodetu solo 26 códigos diferentes de 1.987 finales (1,3%). Como era esperable, relajar la

limitación de bloques aumenta la expresividad: un 25,8% de los aprendices encuentran 23 códigos que no se habían propuesto en Kodetu. Y es importante contrastar que aunque la limitación se aumenta en 3 bloques (5 a 8), la longitud media de código de los aprendices exitosos aumenta en mucho menor medida (4,4 bloques de media en Kodetu frente a 5,1 en KodetuGen2).

El otro nivel de contraste es el 10 (sin correspondencia en Kodetu), que se dejó sin límites en la herramienta experimentada. Hay que tener en cuenta que se llega al nivel 10 tras tres niveles ya de trabajo con estructuras repetitivas, dos de ellos con restricción máxima (niveles 7 y 8) y otro con limitación pero expresividad (nivel 9: 8 bloques máximo, hay soluciones entre 5 y 8). La información de límite de bloques además no es explícita en el inicio del nivel (no aparece un pequeño texto que informa cuántos bloques quedan disponibles). Encontramos tres resultados relevantes:

- A pesar de que el nivel 9 y el 10 son muy similares (el 10 necesita solo un bloque más, de giro inicial, que el 9), las diferencias son muy significativas. En el nivel 9 hay 21 variantes de código de 550 aprendices (3,8%), 47 de 370 en el nivel 10 (12,7%). La solución media es de 5,2 bloques en el nivel 9 frente a 8,9 en el nivel 10.
- Un 8,9% de los aprendices vuelven a utilizar secuencias y desprecian las estructuras repetitivas, para resolver un nivel que en ese caso necesita al menos 18 bloques.
- Un 13,8% adicional de los aprendices utilizan estructuras repetitivas, pero encuentran soluciones indeseables: repetitivas que incluyen el doble de los bloques necesarios, o repetitivas anidadas (que en KodetuGen2 no tienen ningún sentido porque la repetitiva es hasta que se llegue a la meta objetivo, con lo cual no acaba hasta que no acabe el juego).

En segundo lugar, realizamos una revisión de código del resto de niveles donde pueden hacerse comparaciones. Aunque ya hemos visto que las mayores ventajas comparativas de KodetuGen2 pueden ocurrir en los niveles más abstractos, es interesante comparar el nivel 6 con el 7 de Kodetu (el último de los secuenciales). En este nivel el 81,85% de los aprendices de KodetuGen2 encuentran la solución más eficiente por un 70,01% en Kodetu. La longitud media de código en KodetuGen2 es 15,66 por 16,25 en Kodetu.

Los niveles 13 a 15 tienen ya todos los bloques disponibles, las mismas restricciones de longitud y siempre uno de los laberintos de KodetuGen2 está en Kodetu, con lo cual todas las soluciones encontradas de KodetuGen2 son válidas para Kodetu (no al revés). Partiendo de esto, encontramos los siguientes patrones relevantes:

- En el nivel 13 de KodetuGen2 hay 26 soluciones distintas de 70 aprendices, por 156 de 873 en Kodetu. A pesar de esa poca representatividad de soluciones diferentes, hay 6 soluciones (el 23%) que los aprendices de Kodetu no han encontrado.
- En el nivel 14, la longitud de código media de los aprendices con éxito en KodetuGen2 es un poco mayor que la de Kodetu (6,24 frente a 6,20), pero el número medio de pasos recorridos es menor (26,92 para 28,34). De nuevo aquí hay 3 soluciones (de 13) no encontradas en Kodetu (donde hay 85 códigos distintos).
- En el nivel 15 en KodetuGen2, los 6 códigos diferentes encontrados por los 22 aprendices exitosos tienen una longitud media de 6,23 bloques, mientras que los 34 códigos de los 126 de Kodetu necesitan 7,02 bloques. Además, un 68,2% de los aprendices exitosos encuentran soluciones que hacen óptima la longitud de recorrido del astronauta (en uno de los laberintos hay un camino muy rápido si se hace el código adecuado. Solo el 12,7% de los aprendices de Kodetu encuentran soluciones con recorridos de longitud mínima).

6.3. Conclusiones

A lo largo de este capítulo y el anterior hemos expuesto la metodología de nuestro estudio, partiendo del desarrollo de la nueva herramienta KodetuGen2, que soluciona los problemas más significativos encontrados en su antecesora KodetuGen. Hemos detallado cada uno de los 15 niveles de los que consta el juego, con sus diferencias y motivación con respecto a las dos herramientas previas. También hemos indicado las características de la nueva versión de KodetuWin para considerar las necesidades del procesado de datos de nuestro estudio.

Asimismo, hemos comentado el diseño experimental con el que poníamos en marcha esta tercera fase de nuestra investigación con el énfasis en las sesiones de grupos controladas. Tras la fase de captura de datos de KodetuGen2, hemos indicado el pormenorizado proceso con el que hemos validado las sesiones para filtrar las interacciones válidas, identificar los grupos y marcar las sesiones de cara a evitar todo lo posible los casos atípicos que pudieran introducir ruido en el análisis, limitando también las edades objetivo al rango de 8 a 14, con lo que hemos analizado sesiones de 4.743 aprendices que incluyen 2.624.341 interacciones.

Hemos comentado también las limitaciones fundamentales de nuestro estudio. En primer lugar, el tiempo disponible para la prueba es un factor clave y lleva a muchos aprendices a abandonar prematuramente, lo que lleva a pensar que una herramienta que incorpore generalización debería trabajar con un margen de tiempo mayor. Este hecho provoca que la cantidad de muestras de aprendices en los niveles superiores sea menor y tenga menor relevancia estadística. En segundo

lugar, hemos intentado en los grupos controlados que el trabajo sea individual, pero debido a su tamaño y carácter de juego extraescolar puede haber una pequeña influencia de factores externos como compañeros que comparten algunas soluciones. En tercer lugar, consideramos que la heterogeneidad de centros y de cursos escolares es suficiente, pero la muestra de KodetuGen2 es significativamente inferior a la de Kodetu. En cuarto lugar, no hay adaptación gráfica de Kodetu a la edad de los aprendices, lo que puede provocar un posible efecto desmotivador en las edades superiores. Por último, como en cualquier estudio que captura los datos de caracterización de los aprendices por pregunta directa, hay posibilidad de falsedad en esta información, habiéndose validado la información de edad y sexo, filtrando los datos en función de incoherencias.

A pesar de estas limitaciones, hay resultados relevantes de nuestro estudio, que pasamos a enumerar relacionándolos con nuestras preguntas de investigación.

PI_G1. ¿Permite el análisis de las interacciones de un aprendiz en un sistema de estas características entender el desarrollo del PC?

El análisis de las interacciones de un aprendiz con un sistema de estas características **permite entender mejor cómo se desarrolla el PC**. El registro de grano fino nos permite analizar los códigos que los aprendices prueban, cómo estos evolucionan según se enfrentan a los errores y cómo los distintos colectivos responden de maneras diferenciadas. Los indicadores nos permiten analizar aspectos tanto generales como particulares.

Estas herramientas pueden además servir al doble objetivo de **formar y evaluar** a los aprendices. Nuestra propuesta es que se definan cuadros de mando asociados para que profesorado e investigadores tengan acceso a los resultados de la actividad de los aprendices, muy valiosa en un proceso de aprendizaje del PC.

PI_G2. ¿Puede incluirse la competencia de generalización en los retos de programación de las plataformas lúdicas de aprendizaje autónomo de una manera eficaz?

Hemos validado que **puede incluirse la competencia de generalización** en los retos de programación de las plataformas lúdicas de aprendizaje autónomo de una manera eficaz. Recomendamos que el rango de edades en el que se haga sea algo mayor al de herramientas equivalentes sin generalización (a partir de los 11 años).

PI_G3. ¿Es la generalización un aspecto de influencia significativa en el desarrollo del PC? ¿Cómo impacta en el aprendizaje?

Nuestros resultados muestran que esta exposición mejora el aprendizaje y la capacidad de los aprendices de enfrentarse a los problemas de programación de los niveles más abstractos. **La generalización es un aspecto de influencia significativa en el desarrollo del PC**.

PI_G4. ¿Cómo influyen las características personales (sexo, edad, afinidad tecnológica, y conocimiento de programación previo) en las primeras experiencias del PC y en su relación con la generalización?

Hemos contrastado diferentes **influencias de las características personales** (sexo, edad, afinidad tecnológica y conocimiento de programación previo) en las primeras experiencias del PC y en su relación con la generalización. Las edades más apropiadas para la generalización empiezan a los 11-12 años. Los chicos tienen un desempeño algo mejor que las chicas, pero este elemento tiende a igualarse en los retos más abstractos. Los aprendices con conocimiento previo de programación se comportan mejor en todos los casos.

PI_G5. ¿Afecta la incorporación de la generalización a las soluciones encontradas por los aprendices?

Hemos revisado cómo **afecta la incorporación de la generalización a las soluciones** encontradas por los aprendices. En el último nivel secuencial KodetuGen2 mejora a Kodetu en proporción de aprendices que encuentran la solución más eficiente. La longitud media de código en KodetuGen2 es inferior, en los niveles más complejos los aprendices de KodetuGen2 encuentran más diversidad de soluciones y sus códigos se plantean de forma más generalizable.

PI_G6. ¿Qué consideraciones deben tenerse en cuenta para diseñar este tipo de sistemas?

Realizamos una serie de **consideraciones que deben tenerse en cuenta para diseñar** este tipo de sistemas, en base a los resultados de nuestro trabajo:

- Puede incluirse generalización en los juegos de PC añadiendo en un punto no inicial un doble reto que deba afrontarse con un único código.
- Es importante considerar que no siempre la solución plausible del primero de los retos resuelva el segundo, para obligar al aprendiz a que considere los dos casos y realice el proceso mental de generalización.
- Debe estudiarse bien la limitación de bloques por nivel. Salvo en los niveles tutoriales, recomendamos un número suficientemente pequeño como para evitar soluciones mecánicas con pura secuencialidad y suficientemente grande como para permitir la mayor expresividad de código.

En el siguiente y último capítulo de esta investigación, resumiremos todas las conclusiones y enunciaremos las líneas de trabajo futuras.

La vida no debería ser un viaje hasta la tumba con la intención de llegar de manera segura y en un cuerpo bonito y bien conservado, sino llegar derrapando de costado, envuelto en una nube de humo, totalmente usado, totalmente desgastado y gritando: "¡Guau! ¡Menudo viaje!"

Hunter S. Thompson

Capítulo

7

Conclusiones

En el desarrollo de esta tesis doctoral, se han descrito la serie de herramientas de aprendizaje del PC desarrolladas y las metodologías de investigación que han permitido registrar información de miles de aprendices. Con esos datos hemos realizado una serie de análisis, primero sobre mecánicas habituales en las herramientas de PC, aportando después una nueva manera de incorporar la generalización como un elemento significativo del proceso. Con las limitaciones encontradas en la primera versión de la propuesta, desarrollamos una segunda propuesta que nos permitió completar el experimento con suficientes aprendices para finalmente analizar el impacto en una comparativa entre los tres sistemas.

Recogemos en el presente capítulo las conclusiones de esta investigación, resumiendo el proceso llevado a cabo, indicando sus limitaciones, resumiendo los resultados obtenidos, reseñando algunas de las posibles aplicaciones y señalando algunas de las líneas futuras de investigación que planteamos.

7.1. Resumen del proceso de investigación

En el presente documento, hemos tratado de mantener un orden de capítulos con la mayor coherencia posible de cara a la exposición de la investigación realizada. Sin embargo, el orden no ha sido estrictamente el mismo que el mostrado en la exposición de los capítulos desde el punto de vista temporal del proceso de investigación. Resumimos a continuación las etapas temporales en las que hemos realizado este trabajo.

Etapa 1. Desarrollo de Kodetu y establecimiento de hipótesis

Tras años de trabajo en nuestro equipo de investigación con tecnologías aplicadas a la educación y particularmente a escolares de primaria y secundaria, veníamos detectando la importancia de las herramientas del tipo de Scratch o la Hora de

Código en la incorporación del PC como competencia educativa. También éramos conscientes de la relevancia de analizar el comportamiento de los aprendices para diseñar cada vez mejores herramientas para el aprendizaje, especialmente con la consideración de que este concepto todavía era ajeno y externo a la estructura escolar y, por tanto, las herramientas tenían un papel fundamental.

Para ello, definimos nuestro conjunto de hipótesis, detalladas en el capítulo 3 de este documento. Las contrastamos con una revisión superficial de literatura en la que valoramos si estas hipótesis habían sido planteadas o validadas con anterioridad, y si eran oportunas de acuerdo al estado del arte en ese momento.

Con ellas en mente, a lo largo de 2014 evaluamos los sistemas disponibles para realizar este tipo de trabajo. Ante la ausencia aún de tecnologías configurables para experimentar de forma personalizada y controlable, decidimos elaborar nuestra propia plataforma Kodetu. Elegimos Blockly como librería de base, al tratarse de software libre y abierto y permitir un trabajo web multiplataforma y ajeno a instalaciones de ejecutables o *plugins*. Tras una serie de pruebas cerradas, en septiembre de 2014 lanzamos la versión web abierta de Kodetu y empezamos a conversar con centros educativos para incorporar esta actividad en nuestros talleres con escolares.

Etapa 2. Primer experimento

Entre octubre de 2014 y marzo de 2016 mantuvimos abierta la plataforma y fuimos planteando experimentos controlados con grupos de escolares y algunos grupos extraescolares y de adultos. Al mismo tiempo, el acceso público estaba abierto e invitábamos a algunos centros también a que los difundieran para acceso libre desatendido de los aprendices interesados.

En este tiempo diseñamos además la estructura base para realizar variantes de Kodetu y desarrollamos algunas de ellas. Además de la versión de navegación libre que fue incluida en el primer experimento, una serie de otras versiones se realizaron y probaron pero no llegaron a tener cabida en esta investigación: versiones sin límites de bloques por nivel, así como dos variantes con ludificación utilizando sistemas de puntuación y de ranking diferenciados por edades.

Etapa 3. Primer estudio y validación de hipótesis

Entre 2016 y 2017 concretamos nuestro primer conjunto de datos y realizamos el primer estudio de esta investigación, enfocado en las preguntas de investigación definidas. El resultado de este estudio es la base del capítulo 3 de este documento.

Etapa 4. Definición de KodetuGen, segundo experimento y primera revisión de la literatura y herramientas existentes

Coincidiendo en el tiempo con la tercera etapa, realizamos nuestra primera revisión profunda de la literatura, con la metodología detallada en el apéndice D.

Mientras revisábamos estudios relacionados y en base a la experiencia de nuestro propio estudio, junto a la propia experiencia asociada a nuestra docencia de programación en niveles universitarios, observamos la particularidad de la generalización dentro de las competencias del PC y nos planteamos si podría incluirse este proceso mental de una forma más explícita en los retos de Kodetu. Esto nos llevó a diseñar y posteriormente implementar KodetuGen.

Con esa posibilidad en mente, definimos y contrastamos nuestro segundo conjunto de preguntas de investigación, descritas en los capítulos 4 y 5 de este documento. En esas preguntas de investigación ampliábamos las originales y concretábamos requisitos específicos de generalización, variando en ese momento el centro de este trabajo doctoral hacia la siguiente pregunta de fondo: siendo la generalización un elemento relevante en el PC ¿puede incorporarse de una manera efectiva en las herramientas de aprendizaje del PC?

Entre septiembre de 2016 y marzo de 2018, mantuvimos abierta KodetuGen y trabajamos con aprendices en grupos controlados, ignorando ya a los aprendices anónimos y no supervisados.

Etapa 5. Definición de KodetuGen2 y tercer experimento

A finales de 2017 nos dimos cuenta de los errores de diseño de KodetuGen (descritos en el capítulo 4) e, imaginando que todo su conjunto de datos probablemente no sería muy útil para validar nuestras preguntas de investigación, rediseñamos el sistema y desarrollamos KodetuGen2. En marzo de 2018 desactivamos KodetuGen y unas semanas después la sustituimos por su versión mejorada.

Comenzamos en ese momento a contactar con centros educativos para definir sesiones controladas de KodetuGen2, en lo que se convirtió en el tercer y último experimento de este trabajo doctoral, que se prolongó hasta enero de 2019. Cabe reseñar que toda la experimentación realizada en esta investigación cuenta con la aprobación del Comité de Ética de la Universidad de Deusto.

Etapa 6. Segundo estudio, comparativa y validación de hipótesis

En 2019 definimos nuestro definitivo conjunto de datos y realizamos con él el estudio de la plataforma con generalización KodetuGen2 (desarrollado en el capítulo 5) y la comparativa entre las tres versiones, fundamentalmente Kodetu y KodetuGen2 (desarrollada en el capítulo 6).

Con ese análisis validamos el segundo bloque de preguntas de investigación de este trabajo.

Etapa 7. Segunda revisión de la literatura y escritura de tesis

Por último, realizamos una revisión complementaria actualizando la literatura disponible y con la información de todas estas etapas, pasamos a escribir la memoria de tesis doctoral recogida en este documento, que ha sido escrita entre junio de 2019 y abril de 2020.

7.2. Limitaciones de la investigación

Antes de explicar las conclusiones de nuestro estudio, repasamos sus limitaciones más importantes.

La primera de ellas tiene que ver con las limitaciones temporales en el uso de nuestras plataformas en grupos controlados. El tiempo disponible para la prueba es un factor clave y lleva a muchos aprendices a abandonar prematuramente. Deberíamos plantear KodetuGen2 con sesiones de mayor duración ya que hemos constatado que requiere mucha más reflexión y dedicación de la que permite afrontarlo con éxito a la mayoría de los aprendices. Esto provoca que haya pocos aprendices en los niveles superiores, lo que dificulta la validación de los datos obtenidos.

Otra limitación tiene que ver con la limitación de la comunicación entre aprendices durante las pruebas. La manera en la que se realiza la prueba en los grupos experimentales no nos garantiza que no haya influencia de factores exógenos al juego: compañeros que se comentan la solución de un nivel, niños o niñas entusiasmados con su superación de logro que se prestan rápidos a solucionárselo a sus amigos cercanos. Hemos marcado la pauta de que el trabajo sea individual e intentado mantener un ambiente silencioso y de trabajo en las aulas, pero esta actividad se ha planteado como un juego para los aprendices y no se ha tenido el nivel de exigencia que podría asegurar resultados puramente individuales.

La variedad de la muestra de participantes en nuestros experimentos es otra posible limitación de este estudio. Aunque hemos intentado mantener la proporción de centros educativos públicos/privados y hay suficiente variedad de centros y de cursos escolares como para garantizar cierta heterogeneidad en los aprendices sometidos a las pruebas, los centros siempre han participado de forma voluntaria y gratuita en este estudio y eso puede introducir cierto sesgo. Se ha intentado paliar esta situación con un número de aprendices alto, pero quizás necesitaríamos ampliar la muestra de KodetuGen2 para equipararla a la de Kodetu (que actualmente es cuatro veces mayor).

Dadas las restricciones éticas y legales con respecto a la privacidad de los participantes, no podemos asegurar al 100% que todos ellos hayan introducido correctamente su información personal. Los datos de caracterización de los aprendices (edad, sexo, conocimiento previo de programación, afinidad a la tecnología, participaciones previas en Kodetu, etc.) han sido captados por pregunta directa sin supervisión, con lo que hay posibilidad de falsedad en esta información. Hemos podido validar la información de edad y parcialmente la de sexo, y hemos actuado en el filtrado de los datos en función de incoherencias, pero sin garantías de que toda la información sea veraz.

Finalmente, aunque el abandono y la desmotivación son elementos inevitables en cualquier proceso de aprendizaje, ya hemos comentado que habría sido mejorable la adaptación gráfica de los retos de Kodetu y sus versiones a las edades superiores, donde probablemente se haya producido un efecto de sensación de "infantil" que ha influido en los resultados.

7.3. Resultados de la investigación

A pesar de las limitaciones hemos encontrado resultados relevantes en nuestro estudio. En la primera parte, expuesta en el capítulo 3, vemos que los resultados ayudan a entender los primeros pasos de programadores aprendices y pueden servir también de consideración para diseñar mejores herramientas de PC. Revisamos las conclusiones asociadas a nuestras preguntas de investigación:

PI_K1. ¿Cuál es la influencia de las características personales (sexo, edad, afinidad tecnológica y conocimiento de programación previo) en las primeras experiencias del PC?

Encontramos que la edad influye significativamente en el rendimiento en estos desafíos, mejorando según avanza hasta 13-14 años, aunque a partir de ese punto el rendimiento empeora. El sexo influye también en el rendimiento: los chicos resuelven los retos mejor que en las chicas en los primeros niveles, y al hacerse más abstractos con la incorporación de las estructuras combinadas, esta tendencia se invierte. El conocimiento previo de programación no parece ser una ventaja ya que en algunos indicadores su comportamiento es peor; tampoco la afinidad tecnológica es un buen predictor del desempeño. Vemos recomendable adaptar el conjunto de retos si los aprendices tienen edades de educación primaria o primer ciclo de secundaria (por debajo de 14 años). Al respecto del sexo, las plataformas de aprendizaje podrían adaptar la manera en la que suministran ayuda y refuerzo positivo. En los primeros niveles para las chicas; en los retos con estructuras de control complejas, para los chicos.

PI_K2. ¿Cuántas soluciones diferentes encuentran los aprendices en sus primeros retos de programación y qué se puede concluir de estas diferencias?

Hemos visto que a pesar de las fuertes limitaciones que propone Kodetu en los niveles con bloques estructurados, existe una importante diversidad de soluciones que permite identificar la relativa originalidad de los aprendices, así como su habilidad para reutilizar soluciones cuando es posible. La originalidad de código se manifiesta en diferentes aprendices en los niveles secuenciales que en los estructurados. La plataforma podría detectar desviaciones desde los valores más frecuentes, tanto en casos positivos (código diferente pero eficiente) como en los negativos (diversidad causada por ineficiencia de código) y ofrecer sugerencias adecuadas.

PI_K3. ¿Qué variables podrían ayudar a plantear estrategias de personalización y adaptación de este tipo de herramientas de aprendizaje del PC?

Tanto la edad como el sexo permiten plantear personalizaciones de las herramientas, así como la originalidad relativa del código según los aprendices avanzan en los retos. Hemos encontrado dos grandes obstáculos que limitan el progreso de nuestros aprendices: el límite de tiempo y las estructuras de control implicadas en cada reto. Considerando esto, podríamos ajustar dinámicamente el tiempo disponible o el conjunto de retos presentado a cada aprendiz, de acuerdo a sus características iniciales y a la manera en la que va resolviendo los retos anteriores.

A pesar de las limitaciones comentadas en la sección 7.2, consideramos que el volumen de información obtenida es muy significativo, y el preproceso de filtrado minimiza el impacto de estas limitaciones en los resultados finales de nuestro análisis. Para facilitar la colaboración con la comunidad científica, publicamos en 2017 el conjunto de datos completo descrito en el capítulo 3 con acceso abierto a través del *Open Science Framework*: https://osf.io/qjrs9/. Hasta nuestro conocimiento, no existía en la fecha otro estudio que hubiera liberado algún registro detallado con más de un millón de interacciones de forma abierta.

Tras finalizar esta parte de nuestra investigación, Kodetu seguía encontrándose *online* y abierta, disponible en inglés, español y euskera. Sin embargo, nos planteamos la relevancia de variar la plataforma para obtener resultados más concluyentes en alguna de las líneas de interés: la posibilidad de personalizar la plataforma de acuerdo al comportamiento de los aprendices, la incorporación de ludificación y sus posibilidades para ampliar la motivación y el rendimiento, la eliminación de los límites de bloques en todos o algunos de los niveles para aumentar la expresividad del código planteado por los aprendices, etc.

Entre 2016 y 2017 desarrollamos varias versiones prototipo que exploraban algunas de estas opciones (en particular, Kodetu ludificado, Kodetu sin límites y Kodetu con retos personalizables). Pero la dificultad de abordar todas estas

posibilidades en nuestra investigación nos forzó a elegir una de las opciones como la más relevante: la generalización.

Con los resultados de esa última fase de nuestro estudio, expuestos en los capítulos 5 y 6, comentamos las conclusiones a nuestras preguntas de investigación:

PI_G1. ¿Permite el análisis de las interacciones de un aprendiz en un sistema de estas características entender el desarrollo del PC?

El análisis de las interacciones de un aprendiz con un sistema de estas características **permite entender mejor cómo se desarrolla el PC**. El registro profuso y sistemático de las sesiones de juego nos permiten analizar los códigos que los aprendices prueban, cómo estos evolucionan según se enfrentan a los errores y también observar cómo los distintos colectivos responden de maneras diferenciadas, con diversidad de variables que además nos permiten analizar aspectos muy generales (desempeño) y muy particulares (reflexión, miedo al error, adaptación a las circunstancias, tipos de pensamiento para diferente cualidad de reto).

Entendemos además el potencial que estas herramientas tienen de servir al doble objetivo de **formar y evaluar** a los aprendices.

PI_G2. ¿Puede incluirse la competencia de generalización en los retos de programación de las plataformas lúdicas de aprendizaje autónomo de una manera eficaz?

Hemos validado que **puede incluirse la competencia de generalización** en los retos de programación de las plataformas lúdicas de aprendizaje autónomo de una manera eficaz. Tras un intento mejorable (KG) hemos propuesto una manera organizada y sistemática de hacerlo (KG2). Recomendamos además que el rango de edades en el que se haga sea diferente y algo mayor al de herramientas convencionales de PC (a partir de los 11 años, probablemente como una segunda fase tras haber expuesto a los aprendices a experiencias iniciales).

PI_G3. ¿Es la generalización un aspecto de influencia significativa en el desarrollo del PC? ¿Cómo impacta en el aprendizaje?

Nuestros resultados muestran que esta exposición mejora el aprendizaje y la capacidad de los aprendices de enfrentarse a problemas de programación más abstractos. **La generalización es un aspecto de influencia significativa en el desarrollo del PC**, al observar cómo el desempeño en los niveles más abstractos varía sustancialmente cuando los aprendices realizan un entrenamiento previo con un planteamiento generalizado.

PI_G4. ¿Cómo influyen las características personales (sexo, edad, afinidad tecnológica, y conocimiento de programación previo) en las primeras experiencias del PC y en su relación con la generalización?

Hemos contrastado diferentes **influencias de las características personales** (sexo, edad, afinidad tecnológica y conocimiento de programación previo) en las primeras experiencias del PC y en su relación con la generalización. Resumimos las más significativas: las edades más apropiadas para la generalización empiezan a los 11-12 años, donde además se observa un efecto de implicación mayor que a los 12-13, posiblemente debido al cambio de ciclo. Los chicos tienen un desempeño algo mejor que las chicas, pero este elemento tiende a igualarse en los retos más abstractos o menos mecánicos. Los aprendices con conocimiento previo de programación se comportan mejor en todos los casos. La afinidad por la tecnología no es un buen predictor de comportamiento, especialmente en los niveles más complejos de PC.

PI_G5. ¿Afecta la incorporación de la generalización a las soluciones encontradas por los aprendices?

Hemos revisado cómo **afecta la incorporación de la generalización a las soluciones** encontradas por los aprendices. Ya en el último nivel secuencial KodetuGen2 mejora a Kodetu: la solución más eficiente es encontrada por un 11,8% más de los aprendices de KodetuGen2 que los de Kodetu. La longitud media de código en KodetuGen2 es 0,59 unidades inferior. En los niveles con estructuras repetitivas y alternativas, los aprendices de KodetuGen2 encuentran más diversidad de soluciones y sus códigos se plantean de forma más generalizable y resuelven los laberintos en menos pasos.

También hay consideraciones que se integran con la limitación de bloques. En particular, la expresividad aumenta relajando ligeramente el límite de bloques mientras que la longitud de código aumenta en menor medida. Esto contrasta con que, si se eliminan los límites, un porcentaje significativo (8,9%) de los aprendices vuelven a usar secuencias, lo que justifica utilizar límites equilibrados en los procesos de aprendizaje del PC.

PI_G6. ¿Qué consideraciones deben tenerse en cuenta para diseñar este tipo de sistemas?

Realizamos una serie de **consideraciones que deben tenerse en cuenta para diseñar** este tipo de sistemas, en base a los resultados de nuestro trabajo:

- Puede incluirse generalización en las plataformas de PC añadiendo en un punto no inicial un doble reto que deba afrontarse con un único código.
- Es importante considerar que no siempre la solución plausible del primero de los retos resuelva el segundo, para obligar al aprendiz a que considere los dos casos y realice el proceso mental de generalización que le lleve a la solución (y fomente el aprendizaje).

- La progresión de dificultad debe comprobarse de modo experimental registrando el comportamiento de grupos de aprendices en versiones preliminares.
- Debe estudiarse bien la limitación de bloques por nivel. Salvo en los niveles tutoriales, recomendamos un número suficientemente pequeño como para evitar soluciones mecánicas y suficientemente grande como para permitir la mayor expresividad de código.
- Para conseguir la interpretación y evaluación más profunda del aprendizaje de los aprendices es conveniente integrar la herramienta formativa con sistemas que permitan el análisis, con indicadores personalizados para la actividad particular que se propone.
- El marcado de tiempo de los registros debe estudiarse muy cuidadosamente e incluirse de forma duplicada si es necesario (tiempo de navegador y tiempo de base de datos, por ejemplo). Es habitual en los análisis basados en registros de interacción que los tiempos y los umbrales que determinan casos frontera sean importantes en el análisis y la interpretación, y que su corrección obligue a un esfuerzo significativo de depuración de datos.

De acuerdo a todo esto, consideramos que **queda validada la hipótesis** de esta tesis doctoral:

El aprendizaje inicial del PC es un proceso sistematizable en el que la generalización tiene una influencia significativa. Puede modelarse a partir de las interacciones de cada aprendiz en plataformas lúdicas de aprendizaje autónomo, que permiten entender el desarrollo del PC a través de indicadores objetivos.

7.4. Aplicaciones de la investigación

Entendemos que desde los resultados de esta investigación hay dos líneas de aplicabilidad muy claras, que pueden tener en cuenta los desarrolladores de herramientas y contenidos que introducen el PC en la enseñanza obligatoria, de grado y postgrado, educación no formal e informal.

El primer campo de aplicación se refiere a la importancia de **incorporar el registro de grano fino** de la interacción de los aprendices con los sistemas que tratan el PC, así como técnicas y procesos de LA para analizar esos datos. Del estudio de esa información surgen muchas conclusiones que directamente pueden realimentar el flujo de rediseño de los contenidos para permitir una mejora continua, que consideramos especialmente relevante en un campo tan nuevo en la educación y por tanto desconocido como el PC. Por otra parte, esta información puede y también debe ser utilizada por los distintos grupos de investigación para integrarla con otras similares en el medio plazo: mejorar los procesos de aprendizaje, permitir

la personalización eficaz de estos contenidos y generar indicadores que faciliten a los educadores el acceso a la información que los aprendices producen.

El segundo campo de aplicación es la **incorporación explícita de la generalización** en las herramientas de PC. Como hemos mostrado en nuestra investigación, puede incorporarse este aspecto en las mecánicas de las herramientas formativas de PC sin una significativa complejidad, y con las consideraciones de edad y tiempo comentadas, puede permitir un aprendizaje más profundo de un componente esencial como es la abstracción, que es aceptado como elemento clave del PC a la vez que mencionado en muchas ocasiones como una de las características más difíciles de incorporar en el aprendizaje.

7.5. Líneas futuras de trabajo

Tras los resultados de esta investigación, enumeramos una serie de líneas de trabajo futuras que vemos relacionadas con el objeto de nuestro estudio.

Aunque hemos comentado que una de las ventajas de estas plataformas es el doble papel formativo y evaluativo, de cara a las investigaciones nos planteamos definir una **prueba externa** que pueda realizarse tras el paso por Kodetu, de manera que se pueda plantear una evaluación ajena a la herramienta y por tanto completamente independiente de ella. Esta herramienta permitiría contrastar Kodetu o KodetuGen2 con otros materiales alternativos de formación para validar sus diferencias en el aprendizaje del PC y podría basarse en el CTTest validado de Román-González, Pérez-González y Jiménez-Fernandez (2017).

Estamos trabajando ya en colaboración con un grupo de investigación de la Universidad de Tel Aviv en la relación entre PC y creatividad, dentro de uno de los campos que hemos tocado en este estudio y vemos interesantes para completarlo: cómo medir la **originalidad y creatividad en el PC**, y cómo se relaciona con la creatividad en otras áreas.

Hemos comentado con anterioridad que iniciamos el trabajo en algunas versiones de Kodetu ludificado. Creemos que la incorporación de más y mejores **técnicas de ludificación** en la educación en general, y en particular en el aprendizaje del PC, pueden mejorar significativamente el resultado de estas herramientas.

Hemos observado que el juego que da base a la algoritmia de una herramienta de PC (como el laberinto y los movimientos ortogonales de Kodetu) ocurre dentro de un contexto que inevitablemente influye en el aprendizaje. Parece lógico considerar que los recursos o carencias que los aprendices tengan en la **modalidad concreta en la que se trabaja** influirán en el resultado (en el caso de Kodetu, visión espacial o lateralidad). Podrían realizarse otras versiones de esta herramienta basadas en

carreras y saltos, o en sonidos o ritmos, no solo en movimientos en dos dimensiones.

Uno de nuestros resultados experimentales es el abandono que se produce en los aprendices que se enfrentan a la herramienta. Una posibilidad de limitar ese abandono es realizar una versión adaptativa, que vaya detectando la facilidad o dificultad que tiene cada aprendiz para superar un nivel, y pueda ir modificando el planteamiento en función de ello: suministrar pistas, modificar limitaciones de nivel, añadir niveles de refuerzo. En nuestro grupo de investigación se está ya desarrollando un estudio que dará lugar a una tesis doctoral, con el desarrollo de una versión de Kodetu que busca ofrecer rutas de aprendizaje personalizadas con dificultad incremental. Es clave en esta investigación la definición de dificultad en este tipo de juegos y su relación con los conceptos del PC y su aprendizaje. Para ello, estamos estudiando cómo elementos como la configuración espacial del laberinto o el límite temporal afectan al rendimiento de los aprendices. También, en este sentido, nos parece relevante incluir indicadores de algunos de los elementos de analítica en tiempo real, según ocurre el aprendizaje (tanto hacia aprendices como a su profesorado).

Por último, también nos planteamos cómo los hallazgos de esta investigación pueden aplicarse en otras áreas relacionadas con el PC no tan centradas en el pensamiento algorítmico y la programación. Hemos iniciado en nuestro grupo de investigación un estudio que concluirá en otra tesis doctoral que aborda retos no algorítmicos, en particular un sistema ya funcional llamado LEMPEL que plantea retos de búsqueda de patrones de caracteres para conseguir la mejor codificación posible, en base a un sistema de niveles de dificultad progresiva similar al planteado en Kodetu. La investigación intentará aunar elementos de personalización y ludificación con la propia comparativa entre retos algorítmicos y no algorítmicos en el comportamiento de los aprendices, para analizar las relaciones de aprendizaje en facetas diferentes del PC.

7.6. Comentarios finales

En una tarde-noche de abril de 2020 llega ya el momento de escribir la última sección de esta tesis. Hace casi 30 años que me matriculé de doctorado por primera vez. Hoy, a dos habitaciones de distancia está mi hijo Josuan ultimando su trabajo de fin de grado, a una habitación mi hijo Ander proponiendo su proyecto de fin de grado, a la vez que yo escribo estos últimos párrafos. Una tesis doctoral es siempre un largo camino: no sabía que este iba a ser tan largo, pero con todos los pesares, lo agradezco. A los 50 lo he podido paladear mucho mejor. Ese esfuerzo de diseñar herramientas, de contactar con centros escolares, de gestionar a decenas de niños y niñas alborotados. Ese esfuerzo de procesar millones de datos, filtrarlos y pelear con las incoherencias, investigar cual detective a los sospechosos. Ese esfuerzo de crear indicadores para luego desecharlos, de añadir coeficientes para luego cambiarlos. Ese esfuerzo de calcular totales y medias, revisar herramientas estadísticas, visualizar de modos diferentes para encontrar respuestas.

A los 50 he podido redescubrir la ciencia por la que aposté a los 20 y he podido volver a enamorarme de ella. Con tantas personas que reniegan cada día de su trabajo y viven solo esperando a los fines de semana, me considero una persona tan afortunada de disfrutar de todos estos esfuerzos y de no importarme que tantos ocurran en horas no laborales. Me considero tan afortunado de contar con personas que me han entendido y apoyado. Y me considero tan afortunado de haber sido capaz de abordar y finalizar una tarea a largo plazo como es esta partiendo solo de la certeza de ser un buen profesional en las tareas de corto plazo y de dedicaciones concentradas.

No sé si la generalización llegará a estar en unos años incorporada en la mayoría de las herramientas de PC o si nadie volverá a oír hablar de KodetuGen2. Pero, como todo en la ciencia, aquí queda este esfuerzo. Y mientras podamos y nos dejen, seguiremos intentando aportar a esa ciencia. Humilde pero firmemente. Seguiremos intentando cuestionar, experimentar y demostrar. Para seguir haciendo de este mundo un lugar un poquito mejor que el que nos encontramos.

Hay muchos escritos con los que he pensado cerrar este documento. El que hoy me inspira y finalmente elijo es una oración atribuida a Antoine de Saint-Exupéry, autor de El Principito. Conozco pocas inspiraciones que reflejen tanta humildad, generosidad y entrega como ella, y ciertamente el "arte de los pequeños pasos" es el que me ha permitido llegar hasta aquí, y seguramente el que me permitirá llegar allá donde toque a partir de ahora.

"No pido milagros y visiones, Señor, pido la fuerza para la vida diaria.
Enséñame el arte de los pequeños pasos.

Hazme hábil y creativo para notar a tiempo, en la multiplicidad y variedad de lo cotidiano, los conocimientos y experiencias que me atañen personalmente.
Ayúdame a distribuir correctamente mi tiempo: dame la capacidad de distinguir lo esencial de lo secundario.

Te pido fuerza, autocontrol y equilibrio para no dejarme llevar por la vida y organizar sabiamente el curso del día.
Ayúdame a hacer cada cosa de mi presente lo mejor posible, y a reconocer que esta hora es la más importante.

Guárdame de la ingenua creencia de que en la vida todo debe salir bien.
Otórgame la lucidez de reconocer que las dificultades, las derrotas y los fracasos son oportunidades en la vida para crecer y madurar.

Envíame en el momento justo a alguien que tenga el valor de decirme la verdad con amor.
Haz de mí un ser humano que se sienta unido a los que sufren.
Permíteme entregarles en el momento preciso un instante de bondad, con o sin palabras.

No me des lo que yo pido, sino lo que necesito.

En tus manos me entrego.

¡Enséñame el arte de los pequeños pasos!"

Atribuido a Antoine de Saint-Exupéry

Es sorprendentemente complicado esto de mantener algo sencillo.

Mark Zuckerberg

Apéndice

A

Plataformas Kodetu

El propósito de este apéndice no es describir en profundidad la plataforma Kodetu, que incluye decenas de miles de líneas de código JavaScript y más de 5.000 archivos relacionados con la web en producción, sino explicar cuáles son las diferentes versiones de la plataforma, las bases con las que hemos realizado el registro y los modelos de datos utilizados para la información que finalmente ha servido de base para nuestra investigación.

A.1. Versiones de la plataforma

La plataforma Kodetu que hemos utilizado para esta investigación está basada tecnológicamente en Blockly, una librería software desarrollada por Google basada en tecnología web y particularmente en JavaScript. El módulo Blockly que utilizamos de base es *Maze*, que implementa una serie de niveles de laberinto del que hay que encontrar la salida utilizando programación visual basada en bloques.

Kodetu puede alojarse en un servidor web cualquiera ya que toda la lógica se procesa en el navegador cliente. Adicionalmente, hemos desarrollado un servidor de base de datos con una API REST que es al que se conecta el código JavaScript del navegador para registrar los sucesivos datos de entrada en el sistema e interacciones en los niveles.

En base a las modificaciones que hemos ido realizando, algunas de ellas comentadas en este documento, recogemos a continuación el historial de versiones que mantenemos, actualizado en abril de 2020.

Versión 1: Kodetu original. Se desarrolla la primera versión del servidor de datos y su API REST. Se modifica el programa cliente con los 15 niveles diseñados (ver sección 3.2.1). También se añade a este cliente la recogida de información de usuario (datos de clasificación estadística y encuesta de percepción tecnológica) y

se incorpora el primer sistema de registro de interacciones, conectado con el servidor de datos.

Versión 2: Kodetu con saltos (navegación libre de nivel por parte del usuario) y límites (presencia o ausencia de límite de bloques en los niveles superiores). Se diferencian cuatro versiones de cliente Kodetu con cambios de configuración en cada una de ellas: sin saltos de nivel y con límites de bloques por nivel (la original), con saltos y con límites (la incluida junto a la original en el primer experimento -capítulo 3-), sin saltos y sin límites, con saltos y sin límites. No hay cambios en la lógica de niveles.

Versión 3: Kodetu gamificado. Se incorpora al cliente un sistema de puntuación en base a rangos de tiempo y número de bloques, y se añade una interfaz de *feedback* de sensación tras acabar cada nivel (triste a alegre con progresión de 5 opciones). Se integra una tabla de puntuación por edad que clasifica a los aprendices para generar rankings por cada edad. Se modifica la API del servidor para integrar la información de gamificación.

Versión 4: KodetuGen. Se eliminan las características de gamificación y se incorpora la generalización de niveles a partir del nivel 3. Para ello, se modifica sustancialmente el algoritmo original del juego, permitiendo que se reproduzca el código en uno de los laberintos duales o en los dos en forma secuencial; el final de nivel lo marca el éxito de los dos laberintos. Se modifica la API del servidor para recoger la información de reproducción diferenciada de los laberintos.

Versión 5: KodetuGen2. Se mejoran los laberintos y se simplifica la reproducción de los laberintos duales. Se añade funcionalidad de registro de usuario para detectar los aprendices que juegan varias veces y permitirles reanudar la actividad desde el último nivel superado. Se añade también la lógica de grupos para poder gestionar la identificación en bloque en el caso de sesiones de grupo controladas. Se modifica la API del servidor para recoger la información de usuarios y grupos.

Versión 6: Plataforma Kodetu. Se generaliza la tecnología para poder soportar distintas versiones de Kodetu de modo concurrente y se añade la zona web de edición donde los usuarios registrados pueden editar y añadir niveles a un juego de Kodetu o definir juegos Kodetu/KodetuGen2 personalizados. Adicionalmente, se da soporte al registro de usuario para permitir otras actividades diferentes a Kodetu, y se implementa LEMPEL (actividad de compresión de información mediante diccionarios) como primera actividad diferenciada. El sistema de registro y la API del servidor se modifican para dar soporte a estas funcionalidades.

A.2. Proceso de registro de interacción

La base de registro es una función JavaScript que añadimos en el código cliente para enviar al servidor cada uno de los eventos de interacción que hemos identificado para registrar. Como se puede observar en el listado A.1, esta función empaqueta toda la información en un objeto JSON que se envía a través de la API REST al servidor.

```
1.   /** Log user's activity
2.    * @param {user} user id string
3.    * @param {timestamp} timestamp
4.    * @param {level} 1-15
5.    * @param {result} see Maze.RESULT_TYPE
6.    * @param {workspace} XML dump of user's workspace
7.    */
8.   Maze.log = function(user, timestamp, level, result, workspace,
     speed) {
9.     ordenLog++;
10.    var xmlhttp = new XMLHttpRequest();
11.     var code = Blockly.JavaScript.workspaceToCode();
12.     xmlhttp.onreadystatechange = function () {
13.      if(xmlhttp.readyState == 4 && xmlhttp.status == 201) {
14.        logsOk++;
15.      }
16.    }
17.    xmlhttp.open('POST', Maze.API + '/logs', true);
18.    xmlhttp.setRequestHeader("Content-Type", "application/json");
19.    var params = JSON.stringify({
20.      user: parseInt(user),
21.      user_timestamp: timestamp+"",
22.      username: Maze.USER_NAME,
23.      level: level,
24.      levelRef: Maze.LEVEL_REF,
25.      result: result,
26.      workspace: workspace,
27.      code: code,
28.      time: Maze.DURATION,
29.      punt: Maze.CURRENTPOINTS,
30.      age: Maze.AGE,
31.      speed: speed,
32.      challenge: parseInt(Maze.CODE),
33.      challengeCode: Maze.CH_CODE,
34.      challengeUser: parseInt(Maze.CH_USER),
35.      blocks: Blockly.mainWorkspace.getAllBlocks().length,
36.      orden: ordenLog,
37.      timestampMoment: parseInt(moment().valueOf()),
38.      timestampWindows: parseInt(performance.now())
39.    });
40.    logsArray.push(params);
41.    xmlhttp.send(params);
42.  };
```

Listado A.1. Función de registro principal en Kodetu.

En los distintos puntos que se registran (inicio, movimiento de bloque, pulsación del botón de ejecución, etc.) se hacen llamadas a esa función. Podemos ver un ejemplo de código tras la pulsación de ejecución en el listado A.2.

```
1.   Maze.execute = function() {
2.     BlocklyApps.log = [];
3.     BlocklyApps.ticks = 1000;
4.     var code = Blockly.JavaScript.workspaceToCode();
5.     Maze.result = Maze.ResultType.UNSET;
6.     try {
7.       eval(code);
8.       Maze.result = Maze.ResultType.FAILURE;
9.       Maze.log(Maze.USER, Date.now(), Maze.LEVEL, Maze.result,
     'EXECUTE -> FAILURE', Maze.speed);
10.    } catch (e) {
11.      if (e === Infinity) {
12.        Maze.log(Maze.USER, Date.now(), Maze.LEVEL, Maze.result,
     'EXECUTE -> TIMEOUT', Maze.speed);
13.        Maze.result = Maze.ResultType.TIMEOUT;
14.      } else if (e === true) {
15.  (...)
```

Listado A.2. Algunas llamadas de ejemplo a la función de registro.

A.3. Modelo de datos

A continuación, incluimos la parte del modelo de datos más relevante para la información registrada. La entidad esencial está recogida en la tabla log, que recoge una fila por cada evento de interacción (listado A.3). Se observa que en cada interacción se recogen las marcas de tiempo (tanto la del navegador como la de la base de datos) y el estado actual del espacio de trabajo (tanto en formato de bloques XML como en formato de código compuesto por esos bloques).

```
1.   CREATE TABLE `log` (
2.     `id` int(11) NOT NULL AUTO_INCREMENT,
3.     `user` int(11) DEFAULT NULL,
4.     `username` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
5.     `user_timestamp` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT
     NULL,
6.     `challenge` int(11) DEFAULT NULL,
7.     `challenge_code` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT
     NULL,
8.     `challenge_user` int(11) DEFAULT NULL,
9.     `level_ref` int(11) DEFAULT NULL,
10.    `level` int(11) DEFAULT NULL,
11.    `result` int(11) DEFAULT NULL,
12.    `workspace` mediumtext COLLATE utf8mb4_unicode_ci,
13.    `code` mediumtext COLLATE utf8mb4_unicode_ci,
14.    `db_timestamp` datetime DEFAULT NULL ON UPDATE
     CURRENT_TIMESTAMP,
15.    `time` int(11) DEFAULT '0',
16.    `punt` int(11) DEFAULT '0',
17.    `age` int(11) DEFAULT '0',
18.    `speed` int(11) DEFAULT '0',
19.    `blocks` int(11) DEFAULT NULL,
20.    PRIMARY KEY (`id`)
21.  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
     COLLATE=utf8mb4_unicode_ci;
```

Listado A.3. Formato SQL de la tabla principal de la entidad log (registro).

Cada interacción se relaciona con el aprendiz que la realiza, lo que define otra de las entidades principales de nuestro modelo de datos (listado A.4). De cada aprendiz se recoge toda la información de categorización asociada con el experimento (edad, curso escolar, sexo, si ha jugado anteriormente a Kodetu o afinidad tecnológica).

```
1.   CREATE TABLE `users` (
2.     `id` int(11) NOT NULL AUTO_INCREMENT,
3.     `username` varchar(50) COLLATE utf8mb4_unicode_ci NOT NULL,
4.     `password` varchar(500) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
5.     `name` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
6.     `lang` varchar(10) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
7.     `school` varchar(100) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
8.     `course` int(11) DEFAULT NULL,
9.     `plan` varchar(100) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
10.    `age` int(11) DEFAULT NULL,
11.    `gender` varchar(20) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
12.    `country` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
13.    `location` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
14.    `registration_date` datetime DEFAULT NULL,
15.    `avatar` varchar(200) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
16.    `is_active` tinyint(1) DEFAULT NULL,
17.    `tech_skills` int(11) DEFAULT NULL,
18.    `play_kodetu` int(11) DEFAULT NULL,
19.    `tech_like` int(11) DEFAULT NULL,
20.    `db_timestamp` int(11) DEFAULT NULL,
21.    `role` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
22.    `tos` tinyint(1) DEFAULT NULL,
23.    `grupo` int(11) DEFAULT NULL,
24.    PRIMARY KEY (`id`),
25.    UNIQUE KEY `UNIQ_1483A5E9F85E0677` (`username`)
26.  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
     COLLATE=utf8mb4_unicode_ci;
```

Listado A.4. Formato SQL de la tabla principal de la entidad usuario (aprendiz).

Los aprendices en las últimas versiones de Kodetu y KodetuGen2 pueden agruparse para gestionar los grupos experimentales controlados en los que realizamos las pruebas. Esto determina la entidad de grupo, que se relaciona de 1 a N con los aprendices (listado A.5).

```
1.   CREATE TABLE `grupo` (
2.     `id` int(11) NOT NULL AUTO_INCREMENT,
3.     `code` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
4.     `name` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
5.     `type` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
6.     `creator` int(11) DEFAULT NULL,
7.     `username` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
8.     `members` int(11) DEFAULT NULL,
9.     PRIMARY KEY (`id`)
10.  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
     COLLATE=utf8mb4_unicode_ci;
```

Listado A.5. Formato SQL de la tabla principal de la entidad grupo (grupo experimental).

Por último, comentamos dos entidades que no estaban presentes en las primeras versiones de la plataforma. La codificación de los propios niveles de Kodetu estaba

realizada desde el Blockly original en estructuras de datos incluidas en el propio código fuente JavaScript. Para permitir que el sistema permita personalización de retos y que los propios investigadores o educadores puedan diseñar juegos y niveles de juego diferenciados, realizamos una redefinición de las estructuras que permitieran llevar esa información a dos tablas de base de datos. La entidad que recoge cada uno de los juegos que se definen (tiene una fila por cada juego completo: una por Kodetu, otra por KodetuGen) es la mostrada en el listado A.6.

```
1.  CREATE TABLE `challenge` (
2.    `id` int(11) NOT NULL AUTO_INCREMENT,
3.    `code` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
4.    `name` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
5.    `description` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT
    NULL,
6.    `creator` int(11) DEFAULT NULL,
7.    `username` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
8.    `type` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
9.    `levels` int(11) DEFAULT NULL,
10.   `lives` int(11) DEFAULT NULL,
11.   `jumps` tinyint(1) DEFAULT NULL,
12.   `ranking` tinyint(1) DEFAULT NULL,
13.   `creation_date` datetime DEFAULT NULL,
14.   `remixed_from` int(11) DEFAULT NULL,
15.   `image1` longtext COLLATE utf8mb4_unicode_ci,
16.   `published` tinyint(1) DEFAULT NULL,
17.   `image2` longtext COLLATE utf8mb4_unicode_ci,
18.   `type_image` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT
    NULL,
19.   `stars` tinyint(1) DEFAULT NULL,
20.   PRIMARY KEY (`id`)
21. ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;
```

Listado A.6. Formato SQL de la tabla principal de la entidad challenge (reto kodetu).

Y para definir cada uno de los niveles, se utiliza una entidad secundaria relacionada de 1 a N con esta (listado A.7).

```
1.  CREATE TABLE `challenge_level` (
2.    `id` int(11) NOT NULL AUTO_INCREMENT,
3.    `challenge` int(11) DEFAULT NULL,
4.    `level_index` int(11) DEFAULT NULL,
5.    `name` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
6.    `creator` int(11) DEFAULT NULL,
7.    `username` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
8.    `map1` longtext COLLATE utf8mb4_unicode_ci,
9.    `map2` longtext COLLATE utf8mb4_unicode_ci,
10.   `dir1` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
11.   `dir2` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
12.   `block_limit` int(11) DEFAULT NULL,
13.   `run_limit` int(11) DEFAULT NULL,
14.   `time_limit` int(11) DEFAULT NULL,
15.   `extra_life` tinyint(1) DEFAULT NULL,
16.   `repeat_level` tinyint(1) DEFAULT NULL,
17.   `checkpoint` int(11) DEFAULT NULL,
18.   `ranking` tinyint(1) DEFAULT NULL,
19.   `opt_blocks` int(11) DEFAULT NULL,
20.   `opt_time` int(11) DEFAULT NULL,
```

```
21.     `show_ranking` tinyint(1) DEFAULT NULL,
22.     `show_average` tinyint(1) DEFAULT NULL,
23.     `video` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
24.     `maze_if` int(11) DEFAULT NULL,
25.     `maze_if_else` int(11) DEFAULT NULL,
26.     `forever` int(11) DEFAULT NULL,
27.     `answer` longtext COLLATE utf8mb4_unicode_ci,
28.     `remixed_from` int(11) DEFAULT NULL,
29.     `images` int(11) DEFAULT NULL,
30.     `map1editor` longtext COLLATE utf8mb4_unicode_ci,
31.     `map2editor` longtext COLLATE utf8mb4_unicode_ci,
32.     `type` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
33.     PRIMARY KEY (`id`)
34.   ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
      COLLATE=utf8mb4_unicode_ci;
```

Listado A.7. Formato SQL de la tabla principal de la entidad challenge_level (nivel de kodetu).

Tus hijos no son tus hijos
son hijos e hijas de la vida
deseosa de sí misma.
No vienen de ti, sino a través de ti
y aunque estén contigo
no te pertenecen. (...)
Tú eres el arco del cual, tus hijos
como flechas vivas son lanzados.
Deja que la inclinación
en tu mano de arquero
sea para la felicidad.

Kahlil Gibran

Apéndice

B

Herramienta KodetuWin

En este apéndice describimos los aspectos generales de la herramienta KodetuWin, desarrollada dentro del trabajo de investigación de esta tesis y utilizada para diversas fases del proceso de datos.

B.1. Descripción

KodetuWin es una aplicación de escritorio desarrollada en Java SE8 por Andoni Eguíluz. Tiene 43.800 líneas de código fuente en 112 ficheros, con 158 clases, 2.071 métodos y unas 206.000 unidades de código.

Permite leer ficheros de datos de registro, volcados directamente desde el servidor (en formato tabulado o en texto directo de sentencias SQL), y procesarlos de acuerdo a las distintas necesidades que hemos ido encontrando en el desarrollo de esta investigación.

Aunque los usuarios fundamentales son los miembros de nuestro equipo de investigación, unos pocos investigadores con los que colaboramos tienen también acceso a esta aplicación para facilitar el proceso de datos de sus experimentos.

B.2. Funcionalidad

KodetuWin emplea una carpeta en la que se va trabajando con todos los ficheros de datos que importa y luego utiliza. Para ello, existe una opción de configuración donde se puede cambiar la ruta de trabajo, el nombre del conjunto de datos que se quiere usar y algunos otros elementos (figura B.1).

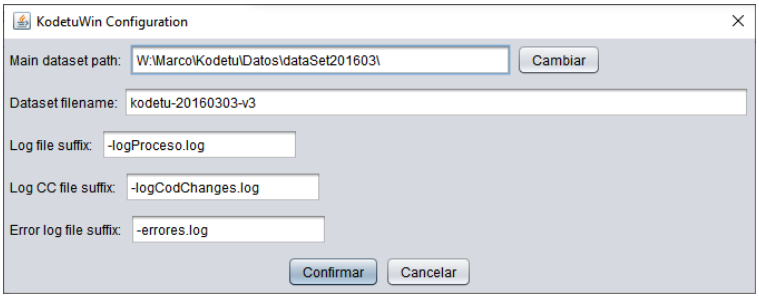


Figura B.1. Cuadro de diálogo de configuración de KodetuWin.

KodetuWin permite cargar un conjunto de datos existente, iniciar uno vacío, o combinar distintos ficheros en el mismo conjunto de datos. La interfaz principal mantiene dos ventanas de trabajo, la primera es de consola para la salida de bajo nivel incluyendo salida estándar y salida de error (figura B.2).

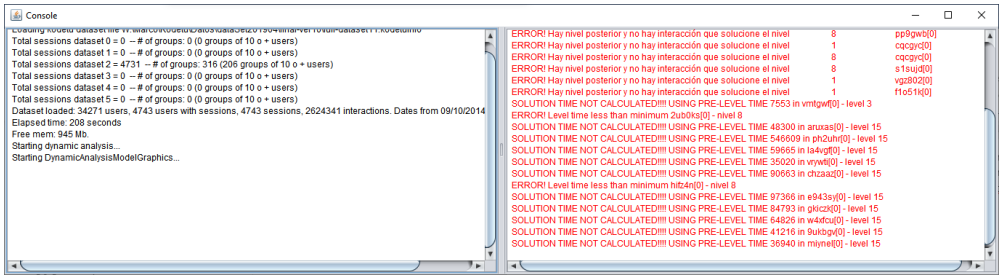


Figura B.2. Ventana de consola de KodetuWin.

La otra ventana es la ventana principal de la aplicación y es la que se utiliza la mayor parte del tiempo (figura B.3).

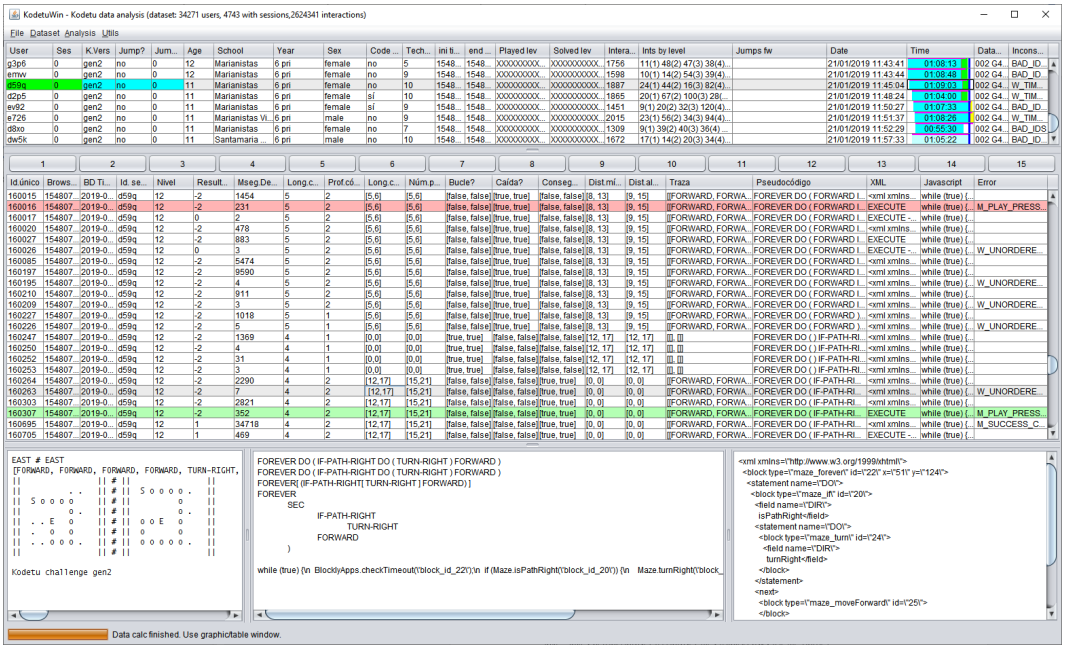


Figura B.3. Ventana principal de KodetuWin.

La ventana principal tiene tres zonas diferenciadas y una línea inferior de progreso y mensajes. El panel superior muestra todas las sesiones del conjunto de datos, una

línea por sesión. Al seleccionar una sesión particular, el panel central funciona como un navegador de sus interacciones, pudiendo verse una interacción por cada línea en el orden en el que ocurrieron, cambio a cambio de código. Las líneas donde el aprendiz intentó la ejecución están coloreadas y hay también una botonera de 15 opciones que permite directamente observar las interacciones del nivel deseado del 1 al 15 (dentro de los que ese aprendiz desarrollara en su experiencia).

El panel inferior muestra el código de la interacción seleccionada, así como una simulación de ejecución de ese código en el laberinto correspondiente. Si la sesión es de KodetuGen o KodetuGen2, se observan los dos laberintos y el camino que ese código trazaría en cada uno. Para ello, KodetuWin tiene implementado un sistema de ejecución que modela cada uno de los bloques de código y no solo permite ejecutarlos como ocurrirían en la aplicación cliente de navegador, sino que se registra una información de perfilado que permite calcular indicadores como número de ejecuciones de cada bloque, código muerto o condiciones comprobadas.

La ventana principal de KodetuWin tiene también una serie de menús desplegables que permiten:

- Operaciones generales de conjunto de datos: crear nuevo, añadir existente, cargar o guardar.
- Lógica dentro de cada conjunto de datos: preproceso de sesiones, calcular versión de Kodetu, dividir sesiones de los usuarios en función de fecha o navegador, comprobación de inconsistencias, comprobación de saltos de nivel, y cálculo de metadatos.
- Análisis de operaciones de conjunto de datos: cálculo de caminos equivalentes, tiempo de nivel, ejecuciones por nivel, cambios por nivel, y una lista de acciones que comentaremos a continuación en la sección de módulos de carga dinámica.

Una serie de facilidades de teclado permiten además navegar con más facilidad por los elementos del conjunto de datos: moverse a sesiones del mismo usuario, buscar una sesión por identificador, reordenar las sesiones por distintos criterios, copiar los datos al portapapeles (para pegarlos en una hoja de cálculo o cualquier otra aplicación), crear y cambiar agrupación dentro del conjunto de datos, etc.

Gestionar los ficheros originales de volcado de datos era una operación que llegaba a tardar unos minutos en Microsoft Excel. Al procesarse desde KodetuWin, se hace un almacenamiento optimizado de los datos en formato binario que permite gestionar de forma interactiva la información sin retrasos significativos.

Otra de las características que hemos programado en KodetuWin es la posibilidad de exportar los datos a una ventana de análisis gráfico (figura B.4). Se puede seleccionar qué parte del conjunto de datos se exporta y partiendo de ella se

genera un conjunto de alrededor de 200 indicadores totalizados por cada uno de los aprendices, detallados en la sección B.4. En esta ventana pueden realizarse una serie de cálculos estadísticos (media, mediana, desviación típica) y exportarse la información para su tratamiento más pormenorizado en una herramienta de tratamiento estadístico (hemos utilizado R para esta investigación). Para el cálculo de medias y medianas pueden filtrarse los datos por versión de Kodetu, edad, sexo, conocimiento de código o afinidad tecnológica, y también segmentarse en dos dimensiones elegibles dentro de los distintos indicadores, para elaborar de forma rápida tablas de datos que se muestran con mapas de calor, y de los que puede obtenerse una visualización rápida que hemos elaborado específicamente para este estudio (figura B.5).

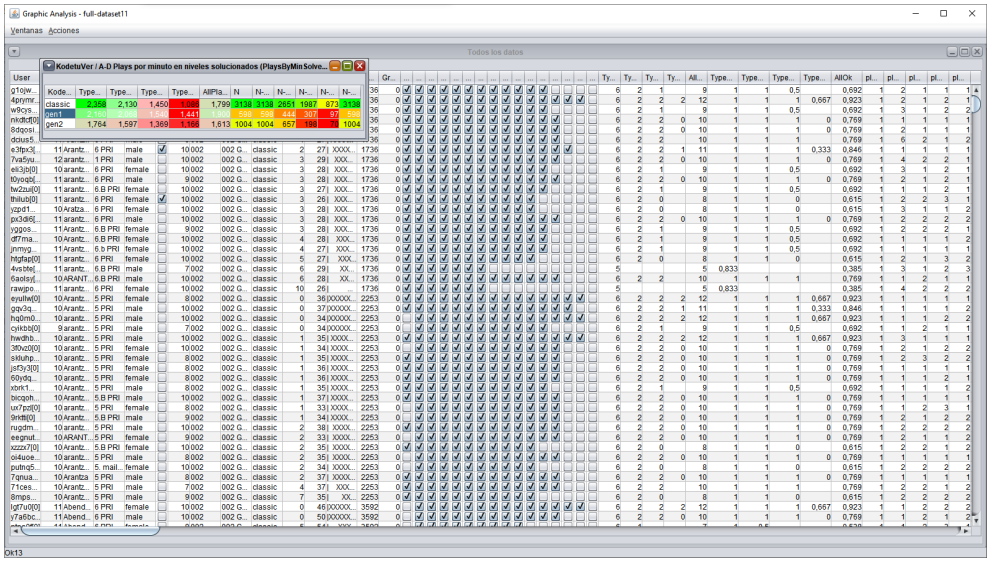


Figura B.4. Ventana de análisis gráfico.

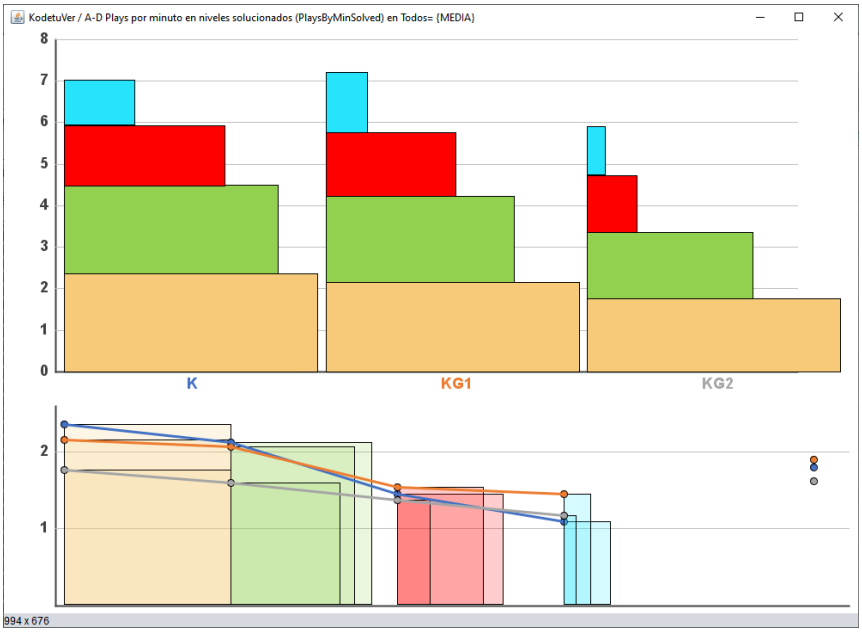


Figura B.5. Ventana gráfica de ejemplo de un análisis determinado (ejecuciones por minuto en función del tipo de Kodetu).

En la fase de análisis de datos, nos ha resultado muy útil poder ir contrastando las diferentes variables para determinar las tendencias y hacer una validación preliminar de las hipótesis, antes de exportar los datos a las herramientas estadísticas desde las que realizar los cálculos definitivos de confianza.

B.3. Módulos de carga dinámica

Cuando tomamos la decisión de desarrollar esta herramienta en Java, nos planteamos ventajas e inconvenientes. La eficiencia de ejecución de manejar los datos en versión local y con un programa compilado fue un factor clave cuando empezamos a darnos cuenta de que la manipulación del conjunto de datos completo podía causar tiempos de espera muy severos. Sin embargo, con un lenguaje compilado teníamos la desventaja de no poder actuar en tiempo real sobre los datos con comandos directos, como puede hacerse en un entorno interpretado como Python o R.

Para paliar esta circunstancia, decidimos desarrollar todo un módulo de análisis de datos que funcionara en un contexto de ejecución de carga dinámica de clases. Java permite hacer esa carga lanzando un cargador diferenciado dentro de la máquina virtual, lo que hace posible que puedan ejecutarse sobre la marcha clases que se acaban de editar mientras el programa ya está funcionando (y todos los datos están cargados), sin necesidad de repetir el proceso de cierre y apertura del sistema.

Así, un importante subconjunto de la funcionalidad de análisis está programado de esa forma. Pongamos un ejemplo: estamos trabajando con el conjunto de datos y nos preguntamos, de los aprendices que han tardado menos de 10 minutos en llegar al nivel 10, cuántos han utilizado un código con alternativas dobles en ese nivel. Podemos programar una clase que realice ese trabajo y cargarla en caliente desde la aplicación, viéndose inmediatamente en consola el resultado de esa ejecución. Para ello, desarrollamos un interfaz que responde a un recorrido del conjunto de datos con una serie de operaciones (listado B.1). Cualquier clase que implemente ese interfaz puede ser lanzada con carga dinámica de la forma descrita.

```
1.    public interface InteractionAnalysis {
2.      /** Llamada al inicio del análisis
3.       * @return       true si se quiere seguir el análisis, false en
      caso contrario (en ese caso se llama solo a endAnalysis)
4.       */
5.      boolean initAnalysis( KodetuDataSet kds );
6.      /** Llamada inicial justo después de initAnalysis. Permite
      reordenar la lista de sesiones por otro criterio, o filtrarla
      quitando sesiones
7.       * @param kds    Dataset sobre el que ser hará el análisis
8.       * @param lKds   Lista de sesiones de ese dataset que se
      trabajarán en el análisis, puede reordenarse o filtrarse
9.       */
```

```
10.    void reorder( KodetuDataSet kds, ArrayList<KodetuSession> lKds
       );
11.    /** Llamada al final del análisis */
12.    void endAnalysis();
13.    /** Llamada al inicio de cada sesión */
14.    void initSession( KodetuSession session, User user );
15.    /** Llamada al final de cada sesión */
16.    void endSession( KodetuSession session );
17.    /** Llamada con la primera interacción en secuencia de cada
       nivel */
18.    void initLevel( KodetuSession session, int nivel, Interaction i,
       boolean isFirstInteractionOfSession );
19.    /** Llamada con toda interacción (incluyendo la primera y la
       última) */
20.    void interaction( KodetuSession session, int nivel, Interaction
       i, boolean isFirstInteractionOfSession, boolean
       isLastInteractionOfSession );
21.    /** Llamada con la última interacción en secuencia de cada nivel
       */
22.    void endLevel( KodetuSession session, int nivel, Interaction i,
       boolean isLastInteractionOfSession );
23.  }
```

Listado B.1. Interfaz de carga dinámica de KodetuWin.

Una serie no exhaustiva de módulos que implementan este interfaz son los siguientes:

- Cálculo de códigos que resuelven el nivel no percibidos por el aprendiz.
- Cálculo de códigos que resuelven el nivel 13 y el 14 en los aprendices que llegan al nivel 14, con conteo de cada una de las casuísticas.
- Comprobación de código de bloques con respecto a código JavaScript.
- Cálculo de mínimos pasos de ejecución y tamaño de código para cada uno de los niveles de laberinto.
- Comprobación de cambios atómicos entre interacciones de los aprendices.
- Filtrados especializados por edad, grupo, desempeño, etc.
- Información totalizada por grupos experimentales.

B.4. Indicadores calculados por aprendiz

A continuación, incluimos el listado de valores que se guardan de cada aprendiz para realizar el proceso estadístico (tabla B.1). KodetuWin permite exportar, calcular estadísticas básicas y mostrar gráficas con cualquier combinación definida sobre estos indicadores.

Campo	Descripción (en inglés)
User	unique string key for user/session
Age	declared age of user
School	declared name of school
ScYear	declared level of actual study year in Spanish form (1-6 PRI is 6-12 years old, 1-4 ESO for 12-16, 1-2 BACH for 16-18)

Campo	Descripción (en inglés)
Sex	declared gender of user. Values: Male/Female
Cod.Know	declared previous coding knowledge: true or false
Tech.Af.	declared affinity to technology: 0 (min) to 10 (max)
DataSet	Number of dataset (always 2)
DataSetGroup	Code of dataset group (002 + space + 'G' + code of group in 4 digits)
KodetuVer	Kodetu version
GroupStartMinute	Minute of session start relative to group
GroupEndMinute	Minute of session end relative to group
GroupTiming	Text expression of session relative to group time (space not, 'X' yes, each character representing 1-minute) - started and ended by vertical bar '|'
GroupTimeLimit	Time limit in the group of this user (seconds)
GroupTimeExcess	Time excess of this user over his/her group (seconds), 0 if not exceeded
GroupLimited	Session finished by user near time limit (true or false)
codeNN	Being NN the level, from 01 to 15. Empty if level NN not solved. If solved: User's code for level NN solution
OkNN	Being NN the level, from 01 to 15. Level solved (true/false)
+NNpre	Being NN the level, from 01 to 15. Code blocks added in level before arriving to a code solving the level (maybe not detected)
-NNpre	Being NN the level, from 01 to 15. Code blocks deleted in level before arriving to a code solving the level (maybe not detected)
*NNpre	Being NN the level, from 01 to 15. Code blocks modified in level before arriving to a code solving the level (maybe not detected). Modifying a block means changing a turn left by turn right or vice versa, or changing an if condition (left/right/forward)
+NNpost	Being NN the level, from 01 to 15. Code blocks added in level from the first code solving the level, to the actual code played by user solving the level. If this and following two fields are cero means user has played the first correct solution obtained
-NNpost	Being NN the level, from 01 to 15. Code blocks deleted in level from the first code solving the level, to the actual code played by user solving the level
*NNpost	Being NN the level, from 01 to 15. Code blocks modified in level from the first code solving the level, to the actual code played by user solving the level
+NNpostSucc	Being NN the level, from 01 to 15. Code blocks added in level after the code solving the level, while playing it
-NNpostSucc	Being NN the level, from 01 to 15. Code blocks deleted in level after the code solving the level, while playing it
*NNpostSucc	Being NN the level, from 01 to 15. Code blocks modified in level after the code solving the level, while playing it
TypeAStartLevel	Number of first level analyzed of type A (Sequences)
TypeAEndLevel	Number of last level analyzed of type A (Sequences)
TypeALevels	Count of levels analyzed of type A (Sequences) = EndLevel - StartLevel + 1
TypeALevelsOk	Count of levels finished of type A (of this session)
TypeBStartLevel	Number of first level analyzed of type B (Loops)
TypeBEndLevel	Number of last level analyzed of type B (Loops)
TypeBLevels	Count of levels analyzed of type B (Loops) = EndLevel - StartLevel + 1
TypeBLevelsOk	Count of levels finished of type B (of this session)
TypeCStartLevel	Number of first level analyzed of type C (Loops+Ifs)
TypeCEndLevel	Number of last level analyzed of type C (Loops+Ifs)
TypeCLevels	Count of levels analyzed of type C (Loops+Ifs) = EndLevel - StartLevel + 1
TypeCLevelsOk	Count of levels finished of type C (of this session)

Campo	Descripción (en inglés)
TypeDStartLevel	Number of first level analyzed of type D (Loops+If-Elses)
TypeDEndLevel	Number of last level analyzed of type D (Loops+If-Elses)
TypeDLevels	Count of levels analyzed of type D (Loops+If-Elses) = EndLevel - StartLevel + 1
TypeDLevelsOk	Count of levels finished of type D (of this session)
AllLevelsOk	All types count of levels finished
TypeAOk	Solved levels of type A. Normalized to [0,1] (levels solved divided by TypeALevels). Blank if no levels of this type played
TypeBOk	Solved levels of type B. Normalized to [0,1] (levels solved divided by TypeBLevels). Blank if no levels of this type played
TypeCOk	Solved levels of type C. Normalized to [0,1] (levels solved divided by TypeCLevels). Blank if no levels of this type played
TypeDOk	Solved levels of type D. Normalized to [0,1] (levels solved divided by TypeDLevels). Blank if no levels of this type played
AllOk	All types solved levels normalized to [0,1] (levels solved divided by level count)
evol.finaNN	Being NN the level, from 01 to 15. Detailed evolution of code constructed by user in level until solved, with following format: #1,#2,#3 (# of block additions, deletions, modifications) ">" (play) "." (finish without play)
playsNN	Being NN the level, from 01 to 15. Number of user plays in level until solved or finished (plays after solution, if so, are not considered)
TypeAPlays	Number of total user plays in type A levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeBPlays	Number of total user plays in type B levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeCPlays	Number of total user plays in type C levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeDPlays	Number of total user plays in type D levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeAPlaysSolved	Average of user plays in type A solved levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeBPlaysSolved	Average of user plays in type B solved levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeCPlaysSolved	Average of user plays in type C solved levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeDPlaysSolved	Average of user plays in type D solved levels (plays after solution, if so, are not considered). Blank if no levels of this type played
TypeAPlaysNotSolved	Average of user plays in type A not-solved levels. Blank if no levels of this type are unsolved
TypeBPlaysNotSolved	Average of user plays in type B not-solved levels. Blank if no levels of this type are unsolved
TypeCPlaysNotSolved	Average of user plays in type C not-solved levels. Blank if no levels of this type are unsolved
TypeDPlaysNotSolved	Average of user plays in type D not-solved levels. Blank if no levels of this type are unsolved
AllPlays	Number of total user plays in all types levels (plays after solution, if so, are not considered)
AllPlaysSolved	Average of user plays in all types solved levels (plays after solution, if so, are not considered)
AllPlaysNotSolved	Average of user plays in all types not-solved levels
T.Bruto NN	Being NN the level, from 01 to 15. Total elapsed time in that level (milliseconds)
T.BetweenLevels NN	Being NN the level, from 01 to 15. Time between last interaction of level NN-1 and last interaction of level NN (empty for 01) (milliseconds)
T.Prelevel NN	Being NN the level, from 01 to 15. Time between last interaction of level NN-1 and first interaction of level NN (milliseconds)
T.Tot.NN	Being NN the level, from 01 to 15. Considered time in that level (milliseconds)
TypeAT.Tot.	Total time used in that type levels (total elapsed time in millis)

Campo	Descripción (en inglés)
TypeATimeByPlay	Time for each play in this type levels (milliseconds): total time divided by number of plays
TypeBT.Tot.	Total time used in that type levels (total elapsed time in millis)
TypeBTimeByPlay	Time for each play in this type levels (milliseconds): total time divided by number of plays
TypeCT.Tot.	Total time used in that type levels (total elapsed time in millis)
TypeCTimeByPlay	Time for each play in this type levels (milliseconds): total time divided by number of plays
TypeDT.Tot.	Total time used in that type levels (total elapsed time in millis)
TypeDTimeByPlay	Time for each play in this type levels (milliseconds): total time divided by number of plays
AllT.Tot.	Total time used in all type levels (total elapsed time in millis)
AllTimeByPlay	Time for each play in all type levels (milliseconds): total time divided by number of plays
TypeAT.Tot.Solved	Average time by solved level in that type levels (millis)
TypeBT.Tot.Solved	Average time by solved level in that type levels (millis)
TypeCT.Tot.Solved	Average time by solved level in that type levels (millis)
TypeDT.Tot.Solved	Average time by solved level in that type levels (millis)
TypeAT.Tot.NotSolved	Average time by not solved level in that type levels (millis)
TypeBT.Tot.NotSolved	Average time by not solved level in that type levels (millis)
TypeCT.Tot.NotSolved	Average time by not solved level in that type levels (millis)
TypeDT.Tot.NotSolved	Average time by not solved level in that type levels (millis)
TypeAPlaysByMinSolved	Average number of plays by minute in solved levels in that type
TypeBPlaysByMinSolved	Average number of plays by minute in solved levels in that type
TypeCPlaysByMinSolved	Average number of plays by minute in solved levels in that type
TypeDPlaysByMinSolved	Average number of plays by minute in solved levels in that type
TypeAPlaysByMinNotSolved	Average number of plays by minute in not solved levels in that type
TypeBPlaysByMinNotSolved	Average number of plays by minute in not solved levels in that type
TypeCPlaysByMinNotSolved	Average number of plays by minute in not solved levels in that type
TypeDPlaysByMinNotSolved	Average number of plays by minute in not solved levels in that type
AllT.Tot.Solved	Average time by solved level in all type levels (millis)
AllT.Tot.NotSolved	Average time by not solved level in all type levels (millis)
AllPlaysByMinSolved	Average number of plays by minute in solved levels in all types
AllPlaysByMinNotSolved	Average number of plays by minute in not solved levels in all types
T.Tot.Tot.Levels	Total elapsed time of user playing from level 2 to last level played
Tot.LevelsJugados	Number of levels played = Number of highest level played (solved or not)
T.Sol.NN	Being NN the level, from 01 to 15. Considered time in that level until solving it (milliseconds)
T.Tot.LevelsSecSol	Total time of solved levels counted from level 2 to last level solved (milliseconds)
LevelMaxSecSol	Not used in sequential kodetus (same than next field)
LevelMaxSol	Max level solved by user
Pr.Ps.Ej.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of steps performed
Pr.Cd.Ej.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of conditions performed
Pr.Lf.Ej.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of turn-left operations performed

Campo	Descripción (en inglés)
Pr.Rg.Ej.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of turn-right operations performed
Pr.Fw.Ej.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of forward operations performed
Pr.It.Ej.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of iterations performed
Pr.Ps.Dd.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of dead blocks (not executed code)
Pr.Cd.Dd.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of dead condition blocks (not executed code)
Pr.Lf.Dd.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of dead turn-left blocks (not executed code)
Pr.Rg.Dd.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of dead turn-right blocks (not executed code)
Pr.Fw.Dd.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of dead forward blocks (not executed code)
Pr.It.Dd.NN	Being NN the level, from 01 to 15. Profiling info of code solution for this level: number of dead loop blocks (not executed code)
Pr.Ps.Tot.	Profiling info of final code for all solved levels: number of steps performed
Pr.Dd.Ps.Tot.	Profiling info of final code for all solved levels: number of dead blocks (not executed code)
Pr2.Ps.Ej.NN	Same for second kodetu maze, if kodetu gen
Pr2.Cd.Ej.NN	Same for second kodetu maze, if kodetu gen
Pr2.Lf.Ej.NN	Same for second kodetu maze, if kodetu gen
Pr2.Rg.Ej.NN	Same for second kodetu maze, if kodetu gen
Pr2.Fw.Ej.NN	Same for second kodetu maze, if kodetu gen
Pr2.It.Ej.NN	Same for second kodetu maze, if kodetu gen
Pr2.Ps.Dd.NN	Same for second kodetu maze, if kodetu gen
Pr2.Cd.Dd.NN	Same for second kodetu maze, if kodetu gen
Pr2.Lf.Dd.NN	Same for second kodetu maze, if kodetu gen
Pr2.Rg.Dd.NN	Same for second kodetu maze, if kodetu gen
Pr2.Fw.Dd.NN	Same for second kodetu maze, if kodetu gen
Pr2.It.Dd.NN	Same for second kodetu maze, if kodetu gen
Pr2.Ps.Tot.	Same for second kodetu maze, if kodetu gen
Pr2.Dd.Ps.Tot.	Same for second kodetu maze, if kodetu gen
TypeAPr.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)
TypeAPr.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeAPr2.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)
TypeAPr2.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeBPr.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)
TypeBPr.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeBPr2.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)
TypeBPr2.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeCPr.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)

Campo	Descripción (en inglés)
TypeCPr.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeCPr2.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)
TypeCPr2.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeDPr.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)
TypeDPr.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeDPr2.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of steps performed (non-solved levels not included)
TypeDPr2.Dd.Ps.Tot.	Profiling info of final code for all solved levels of this type: number of dead blocks (not executed code) (non-solved levels not included)
TypeAPr.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeAPr.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
TypeAPr2.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeAPr2.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
TypeBPr.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeBPr.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
TypeBPr2.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeBPr2.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
TypeCPr.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeCPr.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
TypeCPr2.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeCPr2.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
TypeDPr.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeDPr.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
TypeDPr2.Ps.Tot.Ratio	Profiling info of final code for all solved levels of this type: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
TypeDPr2.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of this type: average of dead blocks by level (not executed code) (non-solved levels not included)
AllPr.Ps.Tot.Ratio	Profiling info of final code for all solved levels of all types: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
AllPr.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of all types: average of dead blocks by level (not executed code) (non-solved levels not included)
AllPr2.Ps.Tot.Ratio	Profiling info of final code for all solved levels of all types: ratio of steps performed over minimum steps for best solution (non-solved lev not included)
AllPr2.Dd.Ps.Tot.Avg	Profiling info of final code for all solved levels of all types: average of dead blocks by level (not executed code) (non-solved levels not included)
numCamb.	Total number of code changes in all solved levels (sum of adds, deletions, and modifications from start to final successful played code)
tSol/camb	Average time for code change (milliseconds) - Relation between number of changes (numCamb.) and total time (T.Sol.NN for all solved levels)
t/camb	Average time for code change (milliseconds) - Relation between number of changes (numCamb.) and total time (T.Sol.NN for all solved levels)

Campo	Descripción (en inglés)
TypeANumCamb.	Total number of code changes in all played levels of this type (sum of adds, deletions, and modifications)
TypeACamb	Average time for code change (milliseconds) - Relation between number of changes and total time of this type
TypeACodeLength	Total number of blocks of final code for solved levels of this type (not-solved levels not included)
TypeBNumCamb.	Total number of code changes in all played levels of this type (sum of adds, deletions, and modifications)
TypeBCamb	Average time for code change (milliseconds) - Relation between number of changes and total time of this type
TypeBCodeLength	Total number of blocks of final code for solved levels of this type (not-solved levels not included)
TypeCNumCamb.	Total number of code changes in all played levels of this type (sum of adds, deletions, and modifications)
TypeCCamb	Average time for code change (milliseconds) - Relation between number of changes and total time of this type
TypeCCodeLength	Total number of blocks of final code for solved levels of this type (not-solved levels not included)
TypeDNumCamb.	Total number of code changes in all played levels of this type (sum of adds, deletions, and modifications)
TypeDCamb	Average time for code change (milliseconds) - Relation between number of changes and total time of this type
TypeDCodeLength	Total number of blocks of final code for solved levels of this type (not-solved levels not included)
AllNumCamb.	Total number of code changes in all played levels of all types (sum of adds, deletions, and modifications)
AllCamb	Average time for code change (milliseconds) - Relation between number of changes and total time of all types
AllCodeLength	Total number of blocks of final code for solved levels of all types (not-solved levels not included)
TypeAExtraChangesAvg	Average of extra code changes (code changes minus minimum code changes for actual code) in solved levels of this type
TypeANotSolvedChanges	Number of code changes in non-solved level of this type
TypeACodeLengthRatio	Ratio: code length / mimimum correct code length for each level of this type
TypeBExtraChangesAvg	Average of extra code changes (code changes minus minimum code changes for actual code) in solved levels of this type
TypeBNotSolvedChanges	Number of code changes in non-solved level of this type
TypeBCodeLengthRatio	Ratio: code length / mimimum correct code length for each level of this type
TypeCExtraChangesAvg	Average of extra code changes (code changes minus minimum code changes for actual code) in solved levels of this type
TypeCNotSolvedChanges	Number of code changes in non-solved level of this type
TypeCCodeLengthRatio	Ratio: code length / mimimum correct code length for each level of this type
TypeDExtraChangesAvg	Average of extra code changes (code changes minus minimum code changes for actual code) in solved levels of this type
TypeDNotSolvedChanges	Number of code changes in non-solved level of this type
TypeDCodeLengthRatio	Ratio: code length / mimimum correct code length for each level of this type
AllExtraChangesAvg	Average of extra code changes (code changes minus minimum code changes for actual code) in solved levels of all types
AllNotSolvedChanges	Number of code changes in non-solved level of all types
AllCodeLengthRatio	Ratio: code length / mimimum correct code length for each level of all types
cancelledPlaysSameCodeNN	Total cancelled plays in each level which finally lead to level solution with same code than cancelled play
cancelledPlaysDifCodeNN	Total cancelled plays in each level which finally lead to level solution with different code
cancelledPlaysNotSolvedNN	Total cancelled plays in each level with finally no level solved

Campo	Descripción (en inglés)
TypeACancelledPlaysSameCode	Total cancelled plays in levels of this type which finally lead to level solution with same code than cancelled play
TypeACancelledPlaysDifCode	Total cancelled plays in levels of this type which finally lead to level solution with different code
TypeACancelledPlaysNotSolved	Total cancelled plays in levels of this type with finally no level solved
TypeBCancelledPlaysSameCode	Total cancelled plays in levels of this type which finally lead to level solution with same code than cancelled play
TypeBCancelledPlaysDifCode	Total cancelled plays in levels of this type which finally lead to level solution with different code
TypeBCancelledPlaysNotSolved	Total cancelled plays in levels of this type with finally no level solved
TypeCCancelledPlaysSameCode	Total cancelled plays in levels of this type which finally lead to level solution with same code than cancelled play
TypeCCancelledPlaysDifCode	Total cancelled plays in levels of this type which finally lead to level solution with different code
TypeCCancelledPlaysNotSolved	Total cancelled plays in levels of this type with finally no level solved
TypeDCancelledPlaysSameCode	Total cancelled plays in levels of this type which finally lead to level solution with same code than cancelled play
TypeDCancelledPlaysDifCode	Total cancelled plays in levels of this type which finally lead to level solution with different code
TypeDCancelledPlaysNotSolved	Total cancelled plays in levels of this type with finally no level solved
AllCancelledPlaysSameCode	Total cancelled plays in levels of all types which finally lead to level solution with same code than cancelled play
AllCancelledPlaysDifCode	Total cancelled plays in levels of all types which finally lead to level solution with different code
AllCancelledPlaysNotSolved	Total cancelled plays in levels of all types with finally no level solved
T.Tot.Levels	Total time of all levels 1 to 15 in millis (sum of T.Tot.NN)
NNCamb	Number of total changes of NN level (sum of +NNpre, -NNpre, *NNpre, +NNpost, -NNpost, *NNpost)
Time	Total session time (minutes)
Ok/t	Solved levels per minute ratio (ok number / T.Tot.Levels)
TypeAOk/t	Solved levels per minute ratio (in levels of this type)
TypeBOk/t	Solved levels per minute ratio (in levels of this type)
TypeCOk/t	Solved levels per minute ratio (in levels of this type)
TypeDOk/t	Solved levels per minute ratio (in levels of this type)
Camb/t	Changes per minute ratio (total changes number / T.Tot.Levels)
TypeACamb/t	Changes per minute ratio (in levels of this type)
TypeBCamb/t	Changes per minute ratio (in levels of this type)
TypeCCamb/t	Changes per minute ratio (in levels of this type)
TypeDCamb/t	Changes per minute ratio (in levels of this type)
posNN	Being NN the level, from 01 to 15. Empty if level NN not solved. If solved: Relative position of user's level NN solution code, as more frequent solution of all users in this level (1 is more frequent solution, 2 second and so on)
%groupNN	Being NN the level, from 01 to 15. Empty if level NN not solved. If solved: Percentage (0.0-1.0] of users with that same solution in level NN
%beforeNN	Being NN the level, from 01 to 15. Empty if level NN not solved. If solved: Percentage (0.0-1.0] of users more frequent solution in level NN than this one

Tabla B.1. Tabla de campos e indicadores calculados de cada aprendiz gestionados en KodetuWin.

Defiéndeme de las fuerzas contrarias,
en el sueño nocturno cuando no soy consciente,
cuando mi camino se hace incierto.
Y no me dejes nunca más, no me dejes nunca más.
Devuélveme a las zonas más altas,
a uno de los reinos de calma.
Es tiempo de escapar de estos ciclos de vidas.
Y no me dejes nunca más, no me dejes nunca más.
¿Por qué los gozos del más profundo afecto
o del anhelo más sutil de pulso
solo son la sombra de la luz?
Recuérdame lo infeliz que me siento
lejos de todas tus leyes.
¿Cómo no malgastar el tiempo que me queda?
Y no me dejes nunca más, no me dejes nunca más.
¿Por qué la paz de ciertos monasterios
o la armonía vibrante de todos mis sentidos
solo son la sombra de la luz?

Franco Battiato. "La sombra de la luz"

Apéndice

C

Traducción del capítulo

Incluimos en este apéndice la traducción a castellano del capítulo completo *"An Evaluation of Open Digital Gaming Platforms for Developing Computational Thinking Skills"* referenciado en el texto de la tesis, particularmente en el segundo capítulo, al respecto de las herramientas de programación basadas en bloques.

C.1. Resumen

Debido a las necesidades empresariales y a la creciente importancia de la tecnología en la sociedad, en los últimos años ha surgido el concepto del PC, especialmente enfocado a su inclusión en la educación obligatoria como un complemento relevante, transversal a las materias tradicionales. Paralelamente iniciativas de diversa índole han ido desarrollando herramientas digitales interactivas para que los aprendices se enfrenten, a través de una serie de actividades comúnmente planteadas como juegos, a este tipo de pensamiento. En este capítulo evaluamos muchas de las plataformas existentes de acceso gratuito y planteamos enfoques pedagógicos, de diseño y de contenido desde los que compararlas.

Palabras clave: pensamiento computacional, aprendizaje, recursos, escuela, videojuegos, lenguajes visuales.

C.2. Introducción

El pensamiento computacional (PC) permite resolver problemas de un modo que una computadora (humana o máquina) puede ejecutar. En solo una década, desde que Wing [1] introdujo el concepto a la comunidad tecnológica, el movimiento ha sido muy intenso, tanto en la comunidad científica como en el mundo educativo, así como en las herramientas y contenido disponible [2]. Gracias a todos estos esfuerzos, empezamos a ver signos de que la situación está cambiando, pero todavía hay muchos retos [3], empezando con la insuficiente definición del concepto del PC y su estructura en los sistemas educativos [4, 5], lo que envuelve múltiples iniciativas de organizaciones nacionales e internacionales.

Considerando lo ubicuo de las computadoras en nuestra Sociedad digital, el PC es una habilidad fundamental no solo para especialistas en computación, sino también para muchos otros profesionales. Todavía es tema de debate cómo de importante y transversal debe ser el PC en educación obligatoria [5]; sin embargo, por razones prácticas, el mundo educativo ha venido prestándole atención creciente. Durante los últimos años, se han desarrollado muchas iniciativas para promover el PC en estudios de educación primaria y secundaria, tanto por el auge social del PC como por la carencia empresarial de profesionales de la computación en el presente y el futuro próximo. La Hora de Código, CodeWeek, o el Scratch Day son algunos de los conocidos eventos globales que promueven cambios en el diseño curricular hacia esta nueva formación digital. La mayoría de estas iniciativas se basan en plataformas online digitales donde los programadores aprendices pueden desarrollar y mejorar sus habilidades de codificación PC a través de juegos.

No es objetivo de este capítulo discutir las ventajas de incorporar la PC en la educación, sino analizar las herramientas, sus usos posibles y sus limitaciones.

Para delimitar el estudio, vemos que hay muchas características comunes entre la mayoría de las herramientas actuales orientadas al desarrollo de habilidades del PC como Code.org, Alice, Scratch o Kodable: son abiertas y gratuitas para su uso general; se orientan fundamentalmente a estudiantes de primaria y secundaria; y hay una característica de juego explícita, que facilita el uso de esos aprendices y utiliza los beneficios demostrados de los juegos en la educación [6].

Es también importante cómo todas estas herramientas buscan evitar que los programadores novatos se enfrenten a la complejidad de la codificación de computadoras basada en lenguajes textuales para mejorar el potencial de aprendizaje [7]. Hay varias maneras de abordar este problema como herramientas narrativas, flujogramas o realizaciones de salida especializada [8]; en este capítulo nos centraremos en las herramientas más habituales, que son las que utilizan programación visual basada en bloques. Estas herramientas se basan en interfases

de usuario basadas en bloques visuales que se mueven y colocan de forma constructiva como un juego de montaje, de hecho habitualmente con la abstracción visual de los puzles con sus piezas y formas de encaje. Estos bloques funcionan como abstracción de los componentes de programación: sentencias, datos, estructuras de control, procedimientos, etc. En consecuencia, limitan considerablemente el conocimiento previo requerido para programar y refuerzan la estructura de programa, eliminando la posibilidad de errores de sintaxis y enfocándose solo en la lógica que existe en la actividad que se quiere plantear.

En este capítulo, realizaremos una revisión de una importante serie de las plataformas existentes con estas características. Hay artículos que comentan algunas [9-11]; nuestra intención es plantear un análisis objetivo, revisando sus posibilidades desde distintas perspectivas. Desde un punto de vista pedagógico, estudiaremos diferentes dimensiones que pueden afectar al proceso de aprendizaje en clase o fuera de ella (como la riqueza de la experiencia interactiva, el tiempo que puede invertirse o la profundidad de la exploración). Desde el punto de vista de juego, la diversión e implicación generadas. Desde un punto de vista del PC, analizaremos qué conceptos implicados en el PC cubre cada plataforma y en qué extensión. Finalmente, analizaremos el grado de adaptación a las características personales de los aprendices de programación, sus habilidades y conocimientos. Consideraremos, entre otros, aspectos como el feedback y la interacción, las posibilidades de registro y el diseño de aprendizaje.

C.3. Revisión de plataformas

Incluimos en nuestro estudio aquellas plataformas accesibles online en julio de 2017, que pueden utilizarse desde el ordenador, móvil o tablet sin licencia comercial de pago, que no requieren hardware o dispositivos adicionales, que están disponibles al menos en idioma inglés, con proyección internacional y abierta, que contienen alguna parte relevante de programación visual y que incorporan alguna característica de juego. En esta sección las describimos en orden alfabético y posteriormente las analizamos en la comparativa de acuerdo a sus características.

C.3.1. Alice (http://www.alice.org/)

Herramienta de creación libre de pequeño juego, animación o recurso interactivo en 3D. Alice es una herramienta de desarrollo de historias animadas interactivas que propone un interfaz de programación visual fuertemente ligado al desarrollo de código orientado a objetos, con conceptos explícitos como métodos, escuchadores de eventos, objetos como personajes, o clases como escenas. En su versión 3 además permite un desarrollo paralelo en el que cada movimiento de bloque explicita el código fuente Java que corresponde. Es de los pocos entornos

orientados a 3D, con lo que posibilita elaborar juegos y animaciones gráficamente impactantes, a la vez que añade un grado de complejidad por la necesidad de elaborar la escena con sus modelos, esqueletos, terrenos o cámaras. Su alianza con Disney, Pixar o EA ha facilitado la inclusión de grafismo muy elaborado, que enriquece la experiencia del proyecto. Creada por la Universidad de Virginia y retomada y mantenida desde entonces por la de Carnegie Mellon en 1997.

C.3.2. App Inventor (http://appinventor.mit.edu/)

Herramienta de creación libre de pequeño juego, animación o recurso interactivo. App Inventor, con un enfoque mucho más cercano al desarrollo profesional que el resto de plataformas analizadas y probablemente la menos cercana a un juego, ya que su objetivo es construir apps para un dispositivo Android. Los usuarios diseñan visualmente el *graphic user interface* (GUI) por un lado y su funcionamiento lógico por otro, simulado en ejecución en el propio dispositivo o en un emulador Android incluído. La dependencia del sistema y del desarrollo para dispositivo móvil hace que la orientación a eventos (sensores, clicks de botón, etc.) esté muy presente en App Inventor y determine el flujo de control. También es importante el diseño de la interfaz de usuario, que tiene una sección específica de la herramienta (Designer) además de la propia de edición de código (Blocks). Creado originalmente por Google en 2010 y retomado por el MIT en 2012 y orientado a un público joven y adulto, no infantil.

C.3.3. Bee-Bot (https://itunes.apple.com/es/app/bee-bot/id500131639?mt=8)

Juego de retos sucesivos para aprender programación para iPad. Bee-Bot es un robot abeja, un juguete físico para niños a partir de 4 años, que tiene también una App gratuita para iPad que puede utilizarse sin el juguete orientado a niños prelectores. Permite que los niños aprendan conceptos de programación sobre primitivas direccionales de movimiento, mientras llevan a su robot virtual a lo largo de una serie de escenarios. Tiene 12 niveles con laberintos de dificultad progresiva. También hay una App de pago con más complejidad para niños a partir de 7 años.

C.3.4. Beetle Blocks (http://beetleblocks.com/)

Herramienta de creación libre de modelo 3D. Beetle Blocks es un entorno de programación visual que permite modelar formas tridimensionales. Como si fuera un lenguaje Logo con su tortuga en 3D, se puede programar el movimiento de un escarabajo virtual para ir generando formas geométricas de todo tipo que se van encadenando con estructuras de control repetitivas y alternativas, y rutinas programables. El lenguaje de bloques está basado en Snap! (analizado posteriormente).

C.3.5. Blockly Games (https://blockly-games.appspot.com/)

Juego de retos sucesivos para aprender programación basado en librería de programación visual generalizable. Blockly (https://developers.google.com/blockly/), desarrollado por Google como *open source* en 2012, realmente es una librería para hacer aplicaciones web basadas en programación visual por bloques (de ahí su nombre). Partiendo de ella se han realizado otras herramientas como App Inventor, Microsoft MakeCode, o OzoBlockly. Analizamos aquí Blockly Games, que es un desarrollo de Google basado en Blockly que propone una serie de juegos educativos para aprender a programar de un modo dirigido y progresivo, en una serie de niveles que se estructuran en siete secciones: puzzle, maze (movimiento para salir de un laberinto con estructuras repetitivas y alternativas), bird (movimiento contínuo en 2D con giros y desplazamiento x-y basado en condiciones y expresiones booleanas), turtle (dibujo estilo Logo con repetitivas sobre ángulos y distancias), movie (movimimento de formas geométricas a partir de un tiempo -fotograma- que cambia de 0 a 100), pond tutor (juego de disparos con ángulos y fuerzas que va introduciendo la programación de texto correspondiente a la visual), y pond (juego de disparos con jugadores controlados por el ordenador a los que vencer). Incorpora además dos desafíos libres que pueden publicarse en Reddit.

La tecnología de desarrollo de Blockly está orientada a la web (en Javascript) pero tiene también soporte nativo para Android y iOS.

C.3.6. Cargo-Bot (https://itunes.apple.com/es/app/cargo-bot/id519690804)

Juego de retos sucesivos para aprender programación. Cargo-Bot es un juego para iPad donde los niños pueden enseñar a un robot a mover cajas dentro de una fábrica utilizando estructuras de programación visuales. El juego contiene 36 puzles y tiene la peculiar característica de haber sido el primer juego desarrollado íntegramente en el propio iPad en base a Codea (un editor de código Lua para iPad). Plantea una codificación con iconos con primitivas de movimiento para la grúa (bajar, subir y mover caja), llamadas a procedimientos para repetir y códigos de color de caja para acciones alternativas.

C.3.7. Code.org (https://code.org/) y Code Studio (https://code.org/educate/gamelab)

Juegos diversos de retos sucesivos para aprender programación, y herramienta de creación libre de pequeño juego, animación o recurso interactivo. Code.org es una organización sin ánimo de lucro creada por el informático de Harvard Hadi Partovi en 2013. Ha sido capaz de dinamizar intensivamente la necesidad de llevar el PC a los jóvenes a través de las actividades promocionadas desde su web y particularmente de la creación de un movimiento llamado "*Hour of Code*" que

propone actividades digitales sencillas en forma de retos de programación realizables en una hora (o pocas horas más) con programación visual de bloques -basada en Blockly-. Al conseguir atraer para la promoción a personalidades del mundo tecnológico como Mark Zuckerberg o Bill Gates, y posteriormente el mismo presidente Barack Obama, el movimiento ha tenido un éxito no solo en USA sino a nivel global, llegando a movilizar a más de 100 millones de niños y adultos que de forma gratuita utilizan sus recursos para "aprender a programar" (*learn to code*). Otro de los aciertos de Code.org ha sido utilizar elementos de videojuegos o películas ya existentes para contextualizar los retos de código. Así ha ido negociando con corporaciones para contar con personajes de Plants vs. Zombies, Flappy Bird, Frozen, Star Wars o Minecraft, lo cual hace mucho más atractivo el proyecto, además de una importante calidad gráfica, de diseño y de lógica. Tras 4 años de horas de código, Code.org y sus partners tienen en la web identificados y accesibles más de 170 actividades diferentes (https://code.org/learn).

Además de las actividades de programación online, Code.org también trabaja el currículum de programación en primaria y secundaria, proporciona materiales formativos y recursos de distintos tipos para uso gratuito de los centros escolares, y promociona actividades formativas para escolares y docentes (con especial atención a mujeres y a *underrepresented minorities*). Para ello ha desarrollado un lenguaje de programación visual de bloques, Code Studio (https://studio.code.org), que al estilo Scratch posibilita la creación libre de recursos multimedia o videojuegos. En esta herramienta se distinguen cuatro tipos de proyectos: Draw something (proyectos de dibujo), App Lab (para simular apps de móvil), Play Lab (juegos o historias sencillas) y Game Lab (juegos más elaborados).

Toda la tecnología desarrollada por Code.org es *open source*. La web incorpora también una sección de profesorado con servicios de gestión para sus grupos de alumnos y una completa página de *dashboard* con control pormenorizado de la evolución de cada estudiante.

C.3.8. Daisy the Dinosaur (http://www.daisythedinosaur.com/)

Juego de retos sucesivos para aprender programación. Daisy the Dinosaur es un pequeño juego para iPad orientado al primer contacto con la programación de los niños más pequeños, que podrán resolver en pocos minutos. Su concepto es muy sencillo: el aprendiz tiene que mover a Daisy la dinosaurio utilizando algunos de los nueve comandos de movimiento con la secuencia adecuada. Cuando acaban los retos, pueden jugar en modo *free-play* para mover libremente al personaje con código libre.

C.3.9. Kodable (https://www.kodable.com/)

Juego de retos sucesivos para aprender programación. Kodable es una de las herramientas más pensadas para profesores y para su uso en clase de una forma estructurada. Contempla una progresión de niveles orientados a toda la edad de educación primaria. Tiene una serie de niveles gratuitos y muchos más de pago con licencias de centro escolar. El planteamiento es de un lenguaje de programación visual basado en iconos que permite mover a un personaje (*Fuzz*) por un laberinto ortogonal recogiendo las estrellas. A las instrucciones de movimiento en las cuatro direcciones se le van añadiendo condicionales usando con casillas de color, bucles con contador, y funciones para repetir varias veces el mismo código. Otra serie de mundos proporcionan complementos formativos de programación conceptuales utilizando juegos, aunque no utilizan propiamente programación visual: un juego estilo tetris para strings y enteros, un *tower defense* para orientación a objetos, etc. Los profesores tienen un *dashboard* con un completo acceso a la información de progreso de sus clases y estudiantes. Además, hay un modo creación para generar niveles personalizados, al que tienen acceso tanto profesor como estudiantes.

C.3.10. Kodetu (http://kodetu.org/)

Juego de retos sucesivos para aprender programación. Kodetu es un juego de resolución de laberintos que permite guiar a un astronauta hacia el objetivo en una estación espacial, utilizando bloques visuales de movimiento y giro y estructuras repetitivas y alternativas. Está basado en Maze de Blockly y propone una secuencia de 15 niveles de dificultad progresiva, hasta llegar a un último laberinto que significa un reto incluso para personas que ya saben programar. Está diseñado para poderse jugar en una hora o dos horas, e incorpora un segundo nivel que trabaja la habilidad de generalización, al tener que resolver dos laberintos utilizando un mismo código. Los profesores pueden generar sus propios grupos y acceder a la información del recorrido realizado por sus estudiantes.

C.3.11. Kodu Game Lab (http://www.kodugamelab.com/)

Herramienta de creación libre de pequeño juego en 3D. Kodu, de nombre original Boku, es un entorno de programación visual desarrollado por Microsoft en 2009, para la consola XBox y SO Windows. Del mismo modo que Alice, el entorno de ejecución es en 3D, pero la orientación de programación es bastante diferente y no es orientada a bloques sino a eventos. El concepto es bastante original con respecto a otras herramientas: el usuario puede modificar el mundo con un sistema visual de edición de suelo y objetos en 3D, y añadir personajes sobre los que se crean reglas (en la forma *when-do*, con lo que llaman en Microsoft *Tile-Based Coding*): si se produce un evento, se ejecuta una acción. Los eventos y reglas están

muy orientadas a un tipo de juego arcade (moverse, disparar, chocar) en un contexto de gravedad fijo. No incorpora estructuras de control ya que el propio sistema de eventos genera una repetición infinita en un bucle de tiempo real, y cada regla (*when*) funciona como alternativa.

C.3.12. Hopscotch (https://www.gethopscotch.com/)

Herramienta de creación libre de pequeño juego, animación o recurso interactivo. Hopscotch, disponible solo para iOS, es un lenguaje de programación visual diseñado específicamente para ser utilizado en dispositivo táctil Apple y muy orientado a juegos sencillos. Aunque la licencia de pago permite personalizar personajes y otras mejoras, en el modo gratuito se puede desarrollar todo el proceso de creación de juego. Hopscotch también tiene una comunidad online para compartir las creaciones, y un reproductor web para que cualquiera pueda jugar desde un navegador a los juegos creados. Es destacable el esfuerzo de Hopscotch en utilizar los recursos de la tablet, tanto para conseguir que todo el sistema de edición pueda realizarse de modo táctil como por incorporar todas las posibilidades del dispositivo en la programación (sensores de inclinación, vibración o acústico).

C.3.13. Lightbot (https://lightbot.com/)

Juego de retos sucesivos para aprender programación. Lightbot es un juego comercial para plataforma móvil (también con versión para Windows y Mac), con una versión de demostración (*code hour*) gratuita que permite jugar los primeros niveles. El planteamiento del juego moderniza los puzles de movimiento clásicos como Robozzle, con retos sucesivos de pequeños laberintos en los que hay que iluminar las casillas azules con las únicas instrucciones de avanzar, girar, saltar y encender. Para repetir se pueden definir varios subprogramas y hacer llamadas recursivas. El juego propone un escalonado de varios niveles de complejidad, y tiene dos apps diferenciadas por dificultad para las franjas de edad de 4-8 y 9+.

C.3.14. Made with Code (https://www.madewithcode.com/)

Juego de retos sucesivos para aprender programación. *Made with Code* es una iniciativa de Google creada en 2014 para animar a las chicas en edad escolar a desarrollar sus primeras experiencias en CT. Incluye mucho material formativo con contenidos textuales y audiovisuales, y propuestas de proyectos y actividades utilizando diversas herramientas. En lo que a nuestro análisis se refiere, tiene una zona específica de proyectos de programación visual (https://www.madewithcode.com/projects/) que propone una serie de actividades cortas con programación visual basada en Blockly. Algunas de estas actividades son

retos de programación con los conceptos básicos (como la basadas en las películas *Inside Out* o *Wonder Woman*), y hay otras más abiertas y creativas que simplemente buscan suministrar herramientas para que las niñas propongan sus propios elementos basados en abstracciones computacionales (como diseñar un patrón de vestuario -*led dress*-, tocar un ritmo musical -*beats*- o crear un mensaje visual en base a sus componentes -*code for equality*-), con una intención clara de mostrar cómo muchas actividades y objetos cotidianos tienen componentes computacionales en su diseño, construcción o uso.

C.3.15. MakeWorld (https://makeworld.eu/)

Herramienta de creación libre de juego de entorno y caminos para definir retos de programación relacionados con otras áreas STEM. Makeworld es una de las pocas plataformas creadas en Europa, dentro además del marco de un proyecto de innovación de la Unión Europea. Se orienta a escolares de primaria, para un entorno de primer aprendizaje de CT. Por ello minimiza los aspectos textuales y propone un interfaz de acciones basado en iconos, con dos niveles: mundos (un reto de programación) e historias (un conjunto de retos enlazados en una secuencia de aprendizaje). A través de estos retos de programación se busca que se trabajen conceptos de otras materias (matemáticas, idiomas, ciencias...) donde se incluyen elementos computacionales: enumeración, secuenciación, identificación, clasificación, ciclos, procesos, sistemas, etc. Los conceptos del PC se limitan al contexto más básico: movimientos en una cuadrícula, repeticiones a través de subprogramas recursivos, y eventos de puntuación para gestionar el progreso. MakeWorld permite tanto resolver mundos planteando un juego de retos, como crear mundos (de ahí su nombre) para ir un paso más allá en el nivel de abstracción. Estos mundos se pueden publicar y compartir en la comunidad social.

C.3.16. Minecraft (https://education.minecraft.net/)

Juego libre de construcción. Minecraft es un juego basado en bloques tridimensionales que permiten crear construcciones en un mundo libre, con una intensa expresión de la creatividad del usuario. Aunque su objetivo inicial era únicamente ser un juego construccionista, ha ido evolucionando y permite incorporar lógicas complejas dentro de los objetos del juego, utilizando desde 2017 un lenguaje de programación visual propio con su entorno de programación visual, *code builder* (con equivalencia en javascript). Puede ser programado con la herramienta de programación visual de Microsoft (MakeCode), o con las existentes de Scratch y Tynker y permite que los elementos del juego cambien y se comporten de acuerdo al código programado. Tras comprar Microsoft el producto original, ha desarrollado toda una línea de educación orientada a plantear problemas y retos de pensamiento computacional y permitiendo compartirlos entre usuarios de la

comunidad. Aunque Minecraft es un producto con licencia comercial, su uso educativo ha ido descendiendo de precio y puede llegar a ser prácticamente gratuito.

C.3.17. Robozzle (http://www.robozzle.com/)

Juego de retos sucesivos para aprender programación. Robozzle, publicado en 2009, es un "juego de puzles social", según su creador Igor Ostrovsky. Plantea un entorno de programación muy básico en un laberinto en el que hay que capturar las estrellas con las únicas instrucciones de avanzar y girar. Para repetir se pueden definir varios subprogramas y hacer llamadas recursivas, y para el concepto de alternativa se utilizan hasta tres colores que determinan si la instrucción se ejecuta o no. El resultado es un pequeño juego de retos para solucionar los puzles que cada nivel plantea. El juego en sí mismo no propone un escalonado de niveles de complejidad (solo están a priori las dificultades básicas del pequeño tutorial), sino que deja de la mano de los propios usuarios la creación de nuevos retos, y su valoración por dificultad y gusto, en una comunidad que ha creado cerca de 10.000 puzles. Permite introducir el PC con un juego sencillo, sin considerar estructuras de control o programas más largos, con muy pocos elementos primitivos, cercano al bajo nivel que hay detrás de los programas.

C.3.18. Scratch (https://scratch.mit.edu/)

Herramienta de creación libre de pequeño juego, animación o recurso interactivo. Scratch es un lenguaje de programación creado por Lifelong Kindergarten Lab del MIT en 2002, visual y con el sistema de edición y ejecución en la nube. Con una importante orientación a la comunidad de usuarios y un enfoque abierto, permite compartir y derivar programas de otros. Debido a su importante historia e implantación, hay multitud de tutoriales para usuarios, profesores y padres. La estructura de componentes de programación diferencia elementos como estructuras de control, eventos, operadores, datos o sensores. Los elementos manipulables por Scratch son imágenes configurables y sonidos, lo que permite programar animaciones y videojuegos en 2D de cierto nivel de complejidad.

Scratch fue una de las primeras herramientas en establecerse y por ello ha influenciado en gran medida a la mayoría de las indicadas en esta revisión. Utiliza la metáfora visual de puzle para las piezas de programación, donde cada bloque tiene una forma que solo puede combinarse con otros bloques compatibles, y un color determinante del tipo de bloque.

C.3.19. ScratchJr (https://www.scratchjr.org/)

Herramienta de creación libre de pequeño juego, animación o recurso. ScratchJr es un lenguaje de programación visual desarrollado como una derivación de Scratch por el mismo departamento del MIT, orientado a usuarios más jóvenes sin habilidades lectoras. El concepto de interfaz cambia (de bloques verticales a bloques horizontales simplificados, reduciendo el número de bloques, y utilizando iconos en lugar de textos) y se orienta a dispositivos móviles, estando disponible de forma gratuita para Android, iOS y Chromebook. Hay también una versión lanzada en colaboración con PBS Kids, que utiliza personajes propios de la serie animada.

C.3.20. Snap! (http://snap.berkeley.edu/)

Herramienta de creación libre de pequeño juego, animación o recurso interactivo. Snap! es un lenguaje de programación visual muy similar a Scratch, inspirado en él en su aspecto y tipo de interacción, pero con una serie de mejoras que lo hacen interesante para un rango mayor de usuarios: se ejecuta en html y javascript con lo que no depende de flash y no limita las plataformas que lo pueden utilizar, permite definir bloques personalizados, gestionar sesiones multiusuario en tiempo real, sprites anidados, generar proyectos como ejecutables, opción de deshacer, funciones de primer nivel, etc. Sin embargo, no hay comunidad de usuarios y proyectos y la documentación disponible para profesorado es muy inferior.

C.3.21. SpriteBox (http://spritebox.com/)

Juego de retos sucesivos para aprender programación. SpriteBox es un juego desarrollado para Hora de Código por la compañía de Lightbot. Tiene un enfoque similar pero sube un poco el nivel del PC al incorporar bucles en lugar de saltos a rutinas. Además incluye un elemento de juego al proponer un juego de plataformas en el que deben superarse retos de programación que afectan al juego (crean plataformas, abren huecos, reconstruyen el escenario). Se acaba en una hora o poco más y plantea niveles de dificultad progresiva en tres mundos consecutivos.

C.3.22. The Foos (http://thefoos.com)

Juego de retos sucesivos para aprender programación. Juego orientado a tablet para niños de preescolar y primaria (no necesita lectura). Se basa en la idea de un juego de plataformas que, además de jugarse como un juego tradicional, permite que el movimiento se configure con una programación de bloques. Nivelado progresivo que va introduciendo las secuencias, repetitivas, eventos y alternativas y acaba con posibilidades de juego libre y creación de niveles personalizados.

La App, creada por la empresa de Pasadena codeSpark en 2014, es comercial pero el uso educativo es gratuito y debe ser gestionado por un docente que

configure la clase, las invitaciones y los dispositivos. Tiene una versión específica de hora de código que es un subconjunto del juego total, y puede ser jugada en el navegador web sin instalación.

C.3.23. Tynker (https://www.tynker.com)

Herramienta de creación libre de pequeño juego, que incluye varios juegos de retos sucesivos para aprender programación. Tynker es un proyecto comercial que tiene una serie de niveles gratuitos, y también un modelo escolar que puede suscribirse sin coste con un conjunto de seis fases (cada una compuesta por una serie de niveles progresivos), y pueden contratarse fases adicionales de pago. El planteamiento del entorno tiene bloques verticales similares a Scratch o Code.org, con mucha variación de contexto y juegos donde elegir. Analizamos los niveles gratuitos y también una sección específica definida para hora de código, Tynker Hour of Code (https://www.tynker.com/hour-of-code) con multitud de niveles diferentes.. Tynker además tiene un entorno libre de programación, que permite editar juegos con un editor a estilo de Scratch, que permite tanto personalizar el escenario y los objetos, como los códigos que estos objetos utilizan con todos los bloques disponibles, lo que lo hace una de las herramientas más completas.

C.3.24. Waterbear (http://waterbearlang.com)

Herramienta de creación libre de pequeño juego, animación o recurso. Waterbear es un lenguaje de programación visual creado por Dethe Elza, profesional canadiense (en un entorno abierto de desarrollo *open source*), inspirado por Scratch pero con un lenguaje desarrollado a poder programar de un modo visual más cercano al lenguaje de programación textual, incorporando elementos como arrays, fechas o diversidad de funciones matemáticas. No hay comunidad de creaciones y el entorno está muy orientado al autoaprendizaje, prácticamente sin material didáctico disponible.

C.3.25. Plataformas no analizadas

Hay muchos otros servicios y productos también orientados al aprendizaje del PC o algunas de sus habilidades, que no hemos considerado en este estudio al no cumplir las condiciones especificadas. En primer lugar, hay toda una serie de juegos que necesitan adquirir dispositivos físicos. Especialmente tienen que ver con robótica y sus derivaciones y, proviniendo de alguna forma de los juguetes clásicos, son un importante nicho de mercado para empresas del sector del juguete educativo y además enriquecen las posibilidades limitadas por una pantalla y un equipo informático "no conectado" al mundo físico. Es el caso de Lego Mindstorms (https://www.lego.com/mindstorms/), que se comercializó ya en 1998 fruto de la

colaboración entre el MIT y la compañía de juguetes de construcción LEGO, para incorporar nuevas piezas robotizadas (motores y sensores de distintos tipos) controlables de forma programada por los niños, utilizando un lenguaje de programación visual que se instala en el ordenador o dispositivo y que permite generar un programa que se transmite a la construcción. Desde 1999 se organiza la FIRST LEGO League, una competición anual internacional con retos científicos y tecnológicos basados en este juego, con más de 200.000 escolares participantes.

En esta misma línea de productos encontramos también muchos otros que han ido apareciendo la última década, como Bee-Bot (https://www.bee-bot.us/), BlocklyProp (https://www.parallax.com/product/program-blocklyprop), Cubelets (http://www.modrobotics.com/cubelets/), Dash the robot (https://www.makewonder.com/), Edison (https://meetedison.com/), Kibo (http://kinderlabrobotics.com/kibo/), Lego WeDo (https://education.lego.com/en-us/elementary/shop/wedo-2), mBot (http://makeblock.com/), microbit (http://microbit.org/), OzoBlockly (http://ozoblockly.com/), Robbo (https://www.robbo.world/), Sphero (http://www.sphero.com/) y muchos otros.

También hay algunos juguetes mixtos físicos/digitales que plantean una parte física sencilla pero necesaria (normalmente piezas a colocar), conectada de algún modo (bluetooth) con una aplicación digital que necesita el "programa" realizado en el mundo físico para superar el reto virtual, parte de lo que se viene denominando *tangible programming*. Es el caso de puzzlets (http://www.digitaldreamlabs.com/), con varios juegos desarrollados orientados a educación primaria, o KIBO (http://kinderlabrobotics.com/kibo/), comercializado en decenas de países.

Mencionamos también una categoría importante representada por GameMaker Studio (https://www.yoyogames.com/gamemaker), GameSalad (http://gamesalad.com/), Stencyl (http://www.stencyl.com/), Unity (https://unity3d.com/) o Unreal (https://www.unrealengine.com/). Estas son herramientas de autor de videojuegos con total o parcial posibilidad de programación visual. Algunas de ellas han sido utilizadas para aprendizaje de programación en aprendices pero no están orientadas como tal al aprendizaje de la programación, ni están específicamente orientadas a niños o jóvenes, ni son comúnmente utilizadas en este tipo de actividad. Sin embargo, el camino que tienen que recorrer para este fin es pequeño, y ya está ocurriendo con alguna de sus iniciativas, como el Kit educativo de Stencyl (http://www.stencyl.com/education/overview/), GameMaker for Education (https://www.yoyogames.com/education) o GameSalad for Education (http://edu.gamesalad.com/), en todos los casos con licencias de pago.

Es también importante reseñar que siguen existiendo entornos de aprendizaje de programación basados en texto, en la línea de los años 90 en los que se empezó

a introducir en las escuelas la programación: Basic (como http://smallbasic.com/) o Logo (por ejemplo MSW Logo: http://www.softronix.com/logo.html).

Un segmento de actividades en incremento para los niveles educativos inferiores son las llamadas "desenchufadas". Por ejemplo Computer Science Unplugged (http://csunplugged.org/), una colección de actividades gratuitas que enseñan PC a través de juegos y puzles con cartas, papel y bolígrafo, cuerdas, y materiales escolares o domésticos, sin considerar herramientas tecnológicas. Otros ejemplos con más enfoque comercial son Hello Ruby (http://www.helloruby.com/), Code Monkey Island (http://codemonkeyplanet.com/) o Robot Turtles (http://www.robotturtles.com/).

También encontramos en el mercado una serie de juegos que no encajan propiamente con el modelo de programación que suele clasificarse como programación visual, pero que sí utilizan conceptos de abstracción, algoritmia y resolución de problemas que fomentan el CT. Es el caso, por ejemplo, de SpaceChem (http://www.zachtronics.com/spacechem/), que propone una serie de puzles en forma de elementos químicos que deben fabricarse con una combinación particular de caminos y operadores sobre átomos y moléculas.

Por último, hay otro gran grupo de herramientas como Code Combat (https://codecombat.com/), Code Hunt (https://www.codehunt.com/), CodeHS (https://codehs.com/), CodinGame (https://www.codingame.com/), Colobot (https://colobot.info/), Minetest (http://www.minetest.net/) o Swift Playgrounds (https://www.apple.com/swift/playgrounds/), juegos de aprendizaje de programación con lenguajes textuales como JavaScript, Java, Python o Swift, sin considerar programación visual. Habitualmente son un recurso muy utilizado para los estudiantes mayores tras haber pasado algunos años por las herramientas de programación visual.

C.4. Análisis de las plataformas

Hemos analizado estas 24 herramientas convirtiéndolas antes en 26, ya que tanto Code.org como Tynker engloban realmente dos planteamientos diferentes que requieren estudio independiente. Exponemos la información considerada en estas plataformas.

C.4.1. Clasificación

La gran mayoría de las herramientas se podrían diferenciar en dos grandes grupos. El primero (un 46,2% de las analizadas) correspondería a un **juego de retos de programación** (ej.: Code.org), en forma de niveles cerrados predefinidos que se van superando, normalmente incorporando nuevas estructuras de programación y aumentando la dificultad de forma progresiva. Las sesiones de uso pueden ser desde unos minutos a unas semanas, donde Code.org está haciendo el esfuerzo

más reseñable por diseñar trayectorias académicas completas con diferentes niveles.

El segundo grupo (42,3%) propone un **lenguaje de programación visual** en sí mismo (ej.: Scratch), con distintos grados de amplitud, con el cual el aprendiz puede desarrollar sus propios programas, en principio de un modo mucho más abierto y muy encajable en una dinámica de aprendizaje basado en proyectos. La mayor parte (27,9%) de los lenguajes se orientan a la programación de un juego 2D, dos (7,7%) a 3D, uno (3,8%) para el modelado 3D y otro para la programación de dispositivo móvil. El componente de juego en este grupo no viene dado realmente por el lenguaje sino por la intención: se pueden realizar juegos, pero también animaciones o historias interactivas, y realmente cualquier aplicación informática que la creatividad del usuario permita (limitada por las primitivas de los lenguajes, que no son de propósito general).

Parece lógico pensar que con los lenguajes podríamos definir las herramientas del primer grupo: es decir, podríamos con Scratch definir un reto de programación (o miles). Sin embargo, las propias estructuras de programación no suelen estar incluidas entre las primitivas del lenguaje (es decir, en Scratch no se puede programar un juego que plantee un reto de programación, salvo con mucho esfuerzo y personalización). Además, las plataformas que proponen retos pueden detectar de forma automática la superación de cada nivel, dar feedback en caso de error, y ofrecer al usuario la navegación al siguiente. En los lenguajes de programación se pueden proponer retos, pero su evaluación y secuenciación son ajenas al sistema. Por eso de momento son dos tipos de herramientas diferenciadas, aunque parece lógico que las tecnologías van a seguir acercando la posibilidad de integrar ambas (como hacen de similar manera Code.org y Tynker).

De momento, sí hay dos herramientas de las analizadas (MakeWorld y Robozzle, un 7,7%) que permiten hacer las dos cosas: los usuarios pueden resolver retos ya planteados, y también pueden crear nuevos retos de código y publicarlos para que otros usuarios los resuelvan. Podríamos hablar entonces de un tercer grupo de herramientas de **creación y solución de retos de programación**.

Una última cuarta categoría representada por Minecraft (3,8%) es la de un **videojuego que incorpora programación visual en sus mecánicas**. El objetivo principal del juego no es la programación visual (de hecho, en Minecraft ha aparecido tras años en los que se podía interactuar solo con lenguajes textuales), pero lo incorpora y permite trabajar aspectos del PC.

C.4.2. Características generales

En la tabla C.1 podemos observar las características generales de las 26 herramientas analizadas. *Type* hace referencia a la clasificación ya comentada. Se indica el país de creación, año de lanzamiento y número de idiomas. El número de

usuarios se ha indicado en los casos en los que se ha encontrado una referencia aproximada de cierta confianza.

Juego	Tipo	País	Año	#Idi.	#Usu.	Tecnología						
						Web	Win	Mac	Linux	Android	iOS	ChromeOS
Alice	LENG	USA	1998	23		--	X	X	X	-	-	-
App Inventor	LENG	USA	2010	11	7M	X	-	-	-	-	-	-
Bee-Bot	RETO	USA	2012	1	300m	-	-	-	-	-	X	-
Beetle Blocks	LENG	USA	2014	39		X	-	-	-	-	-	-
Blockly Games	RETO	USA	2012	49		X	-	-	-	-	-	-
Cargo-Bot	RETO	AUS	2012	1	1M	-	-	-	-	-	X	-
Code.org - Courses	RETO	USA	2011	51	430M	X	-	-	-	-	-	-
Code.org - Code Studio	LENG	USA	2014	51	20M	X	-	-	-	-	-	-
Daisy the Dinosaur	RETO	USA	2013	1		-	-	-	-	-	X	-
Kodable	RETO	USA	2012	1	>1M	X	-	-	-	-	X	-
Kodetu	RETO	ESP	2014	3	10m	X	-	-	-	-	-	-
Kodu Game Lab	LENG	USA	2009	22	2,5M	-	X	-	-	-	-	-
Hopscotch	LENG	USA	2012	3		-	-	-	-	-	X	-
Lightbot	RETO	CAN	2008	28	7M	X*	X	X	-	X	X	-
Made with Code	RETO	USA	2014	1		X	-	-	-	-	-	-
MakeWorld	CREA	ESP	2016	6	2m	X	-	-	-	-	-	-
Minecraft	JUEGO	SWE	2011	11	130M	-	X	X	-	X**	X**	-
Robozzle	CREA	USA	2009	1	130m	X*	-	-	-	X	X	-
Scratch	LENG	USA	2002	72	20M	X*	X	X	X	-	-	-
ScratchJr	LENG	USA	2014	7		-	-	-	-	X	X	X
Snap!	LENG	USA	2011	39		X	-	-	-	-	-	-
SpriteBox	RETO	CAN	2016	2		X*	-	-	-	X	X	-
The Foos	RETO	USA	2014	17	4M	X	-	-	-	X	X	-
Tynker	LENG	USA	2013	1	50M	X	-	-	-	X	X	-
Tynker - Activities	RETO	USA	2013	1	50M	X	-	-	-	X	X	-
Waterbear	LENG	CAN	2011	1		X	-	-	-	-	-	-

Tabla C.1. Características generales (* significa que el navegador necesita plugin flash, silverlight en el caso de Robozzle. ** significa que Minecraft tiene apps comerciales para Android y iOS).

Es muy significativa la hegemonía de USA en este tipo de herramientas: casi tres cuartos (73,1%) de las plataformas se han desarrollado allí, lo cual es coherente con la dominancia en general en los servicios de internet, y también puede responder a la importante campaña por el interés del aprendizaje básico de computación en niveles escolares que lidera USA desde hace una década (se observa que tras Alice y Scratch han ido surgiendo herramientas de manera contínua). Canadá le sigue con un 11,5%, lo mismo que toda Europa, y Australia tiene su plataforma con Cargo-Bot.

Las más antiguas son Alice y Scratch, lo cual explica su influencia y recalca la importancia de las universidades americanas (MIT y Carnegie Mellon) en el campo de la programación visual. Los datos de implantación hablan de las más extendidas: Code.org, Tynker (a pesar de que solo se encuentra en inglés) y Scratch, aunque no hemos encontrado datos de usuarios de algunas significativas como Alice, ScratchJr o Blockly. Aparte está la gran implantación de Minecraft como juego constructivista, sin datos específicos de cuántas personas están utilizando sus posibilidades de programación visual. El mundo web es claramente clave para la utilización masiva, quedando los sistemas que solo funcionan en tablets más centradas en apuestas comerciales de pago, con el iPad con un lugar reseñable debido a su implantación en el mundo escolar en algunos países (como USA).

Para los datos sociales mostrados en la tabla C.2 hemos organizado las plataformas ya por su tipo, porque como se observa las opciones sociales tienen gran correlación con el enfoque de la herramienta. En las que planten retos de programación no hay opciones sociales, salvo las excepciones de Code.org, Blockly Games y The Foos que permiten subir a internet las creaciones realizadas en algunos niveles, que coinciden en ser los que plantean un "modo libre" (lo cual funciona de hecho como una excepción al resto de niveles, donde el reto tiene una solución que se consigue o no). Al contrario, en los lenguajes visuales de programación lo habitual es incorporar una comunidad adicional al propio lenguaje (lo hacen el 69%), que al menos permite subir los proyectos deseados y compartirlos ("share") y, en algunos casos, más opciones sociales como el "like" a programas de otros usuarios, el "report" para denunciar programas inadecuados, el "fav" para marcar favoritos, "follow" para seguir a otro usuario, o el "comm" para comentar con texto libre la creación compartida. En este sentido, las dos herramientas de creación y solución funcionan del mismo modo que los lenguajes visuales, con la particularidad de Robozzle, que es la única que incorpora el "dislike" y la posibilidad de evaluar de 1 a 5 los puzles de otros usuarios.

Tipo	Juego	Tiene comunidad	Permite upload	Opciones sociales	Remix
Retos	Bee-Bot	-	-	-	-
	Blockly Games	limit.	X*	compartir*	limit.
	Cargo-Bot	-	-	-	-
	Code.org - Courses	-	X*	compartir*	-
	Daisy the Dinosaur	-	-	-	-
	Kodable	-	-	-	-
	Kodetu	-	-	-	-
	Lightbot	-	-	-	-
	Made with Code	-	-	-	-
	SpriteBox	-	-	-	-
	The Foos	-	X*	like, compartir*	-
	Tynker - Activities	-	-	-	-

Tipo	Juego	Tiene comunidad	Permite upload	Opciones sociales	Remix
Lenguajes de programación visual	Alice	-	-	-	exp
	App Inventor	X	-	-	exp
	Beetle Blocks	X	X	fav	X
	Code.org - Code Studio	X	X	compartir	X
	Hopscotch	X	X	like / compartir	X
	Kodu Game Lab	X	X	like / compartir / report	X
	Scratch	X	X	fav / like / report / comm / seguir	X
	ScratchJr	-	-	-	-
	Snap!	-	X	-	X
	Tynker	X	X	like / report / compartir	X
	Waterbear	-	-	-	exp
Creación y juego	MakeWorld	X	X	like / seguir / compartir / comm	X
	Robozzle	X	X	like / dislike / evaluar	-
Juego	Minecraft	X	-	-	X

Tabla C.2. Características sociales (* = solo en algunos niveles, exp = remezcla desde fichero exportado).

C.4.3. Aspectos pedagógicos

Abordamos a continuación información relacionada con el uso en clase de estas herramientas. En primer lugar, es fundamental la edad objetivo, que se muestra en la tabla C.3 donde aparecen las edades recomendadas de las distintas plataformas, marcadas de acuerdo a las indicaciones de las compañías desarrolladoras, las opiniones de la comunidad educativa y las propias características de las herramientas. Por una parte vemos que hay herramientas diversas para todas las edades, lo cual es una buena noticia para la comunidad educativa. Por otra parte observándolas por tipos, como era lógico esperar, las plataformas de retos están enfocadas a edades inferiores (edad media 8,1); suben las de creación y solución de retos (9,8); y son mayores los lenguajes para definir juegos (12,4), Este esquema corresponde al escalonado que sería deseable si queremos diseñar el proceso educativo longitudinal con estas herramientas: empezar con una plataforma de retos con los objetivos y el camino marcado, seguir con una de creación de retos en base a propuestas realizadas por el docente de una forma estructurada y guiada, y acabar con un lenguaje visual de propósito más general, en un entorno de aprendizaje basado en proyectos y con libertad de elección personal del aprendiz para la definición y realización de los proyectos.

Tipo	Juego	Edades recomendadas para jugar							
		<5	6-7	8-9	10-11	12-13	14-15	16-17	>18
RETO	Bee-Bot	X	-						
	Blockly Games			X	X	X	X	-	-

Tipo	Juego	Edades recomendadas para jugar							
		<5	6-7	8-9	10-11	12-13	14-15	16-17	>18
	Cargo-Bot	-	X	X	X	-			
	Code.org - Courses		X	X	X	X	-	-	-
	Daisy the Dinosaur	X	X	-					
	Kodable	X	X	X	-	-	-		
	Kodetu			X	X	X	X	-	-
	Lightbot	X	X	X	X	-			
	Made with Code	-	X	X	X	X	X	-	
	SpriteBox	X	X	X	X	-			
	The Foos	X	X	-	-				
	Tynker - Activities			X	X	X	X	-	-
CREA	MakeWorld	-	X	X	X	X	X	-	
	Robozzle		X	X	X	X	-	-	-
LENG	Alice				-	-	X	X	X
	App Inventor						-	X	X
	Beetle Blocks					X	X	X	-
	Code.org - Code Studio				X	X	X	-	-
	Hopscotch			X	X	X	-	-	
	Kodu Game Lab	-	X	X	X	X	-	-	
	Scratch			X	X	X	X	-	-
	ScratchJr	X	X	-	-				
	Snap!			X	X	X	X	X	X
	Tynker			X	X	X	X	-	-
	Waterbear				-	X	X	X	-
JUEGO	Minecraft						-	X	X

Tabla C.3. Edades recomendadas (X = recomendada, - = viable, blanco = no recomendada).

En la tabla C.4 se muestran aspectos de sencillez de instalación y uso (valorados de 0 a 3 de acuerdo a una rúbrica común), riqueza de interacción (también de 0 a 3), rangos de tiempos estimados de dedicación basados en el material disponible y la complejidad y profundidad permitida por cada sistema, y material disponible para profesorado (A-D).

Tipo	Juego	Simp. instal. [1]	Datos personales solicitados	Usabilidad [2]	Riqueza interacción [3]	Rango de tiempo de dedic. aprendiz	Rango de tiempo de prep. prof.	Material disponible prof. [4]
CHAL	Bee-Bot	0	no	0	0	1h-10h	1h-5h	B
	Blockly Games	0	no	0	0	1h-15d	5h-20h	C
	Cargo-Bot	0	no	1	1	1h-5d	1h-10h	A
	Code.org - Courses	0	email (opcional)	0	1	1h-2m	5h-50h	D
	Daisy the Dinosaur	0	no	0	1	1h-4h	1h-5h	B

Tipo	Juego	Simp. instal	Datos personales solicitados	Usabilidad [2]	Riqueza interacción	Rango de tiempo de dedic	Rango de tiempo de prep	Material disponible prof. [4]
	Kodable	0	no	0	1	1h-2m	5h-50h	D
	Kodetu	0	sexo,edad,escuela	0	0	1h-5h	1h-5h	A
	Lightbot	2	no	0	1	1h-3h	1h-5h	C
	Made with Code	0	no	0	1	1h-20h	1h-20h	C
	SpriteBox	0	no	0	1	1h-3h	1h-5h	C
	The Foos	2	email	0	1	1h-30d	5h-30h	C
	Tynker - Activities	0	email (opc.)	0	1	1h-1y	2h-10h	C
CREA	MakeWorld	1	sexo,país,edad,em	1	2	1h-2y	5h-40h	D
	Robozzle	0	email (opc.)	1	1	1h-5d	2h-10h	B
LANG	Alice	2	no	3	3	3d-4y	20h-50h	D
	App Inventor	3	cuenta google	3	3	3d-4y	20h-80h	D
	Beetle Blocks	0	no	1	2	2d-1y	5h-40h	B
	Code.org/Code Studio	0	em,edad,sexo(opc)	0	2	5d-2y	20h-50h	D
	Hopscotch	0	em (opc.)	2	2	5d-2y	20h-50h	C
	Kodu Game Lab	2	no	2	3	5d-2y	10h-50h	C
	Scratch	1	edad,sexo,país,em	2	2	5d-2y	20h-50h	D
	ScratchJr	2	no	2	1	1d-2m	5h-10h	D
	Snap!	0	edad,em	2	2	5d-2y	20h-50h	C
	Tynker	0	em	1	2	5d-2y	5h-50h	C
	Waterbear	0	no	0	2	5d-2y	20h-50h	A
GAME	Minecraft	3	em	3	3	5d-2y	20h-80h	D

Tabla C.4. Otros aspectos pedagógicos. [1], [2], [3] evaluados en rango 0-3, 0 más simple 3 más complejo. [4] valorado A-D, de sin material (A) a material muy variado y multilingüe (D).

Se puede observar la amplitud de los rangos de dedicación, que en el caso de herramientas generales de programación como Scratch o Tynker pueden oscilar de unos pocos días a algunos cursos (multitud de centros educativos utilizan estas herramientas a lo largo de varios cursos, si bien la dedicación no suele ser contínua). Se ve también que los sistemas de reto tienen un planteamiento temporal mucho menos ambicioso, salvo en las más elaboradas en niveles planteados como Tynker o Code.org.

C.4.4. Implicación (*engagement*)

No hay expresiones únicas ni universales de la diversión o la implicación que una actividad digital es capaz de producir en un usuario. Cada persona tiene sus gustos, preferencias y aprendizajes; además, en el entorno escolar la manera en la que una actividad se propone influye mucho en su recepción. No es lo mismo que un niño/a elija libremente un juego al que le apetece jugar en el momento en que desea, a que le invite a hacerlo su profesor/a, más o menos obligatoriamente, dentro de una clase. Ese es un factor que viene influyendo desde el inicio en todos los juegos

educativos. Pero en cualquier caso hemos hecho el ejercicio de intentar objetivar la "diversión" que proponen potencialmente cada una de nuestras 26 herramientas, diferenciando algunas de las dimensiones clásicas que influyen en la experiencia del usuario cuando se trata de videojuegos y valorando cada una de ellas entre 0 (mínimo) y 3 (máximo). Esto nos da un promedio de Engagement que es el que se muestra ordenadamente en la tabla C.5.

Juego	Tipo	Sen	Fan	Nar	Ret	Com	Des	Exp	Ent	His	Implicación
Minecraft	GAME	3	3	3	3	3	3	3	3	2	**2,89**
Code.org - Code Studio	LANG	2	3	2	3	2	2	3	3	3	**2,56**
Tynker	LANG	2	3	2	3	2	2	3	3	3	**2,56**
Scratch	LANG	3	3	2	3	0	2	3	3	3	**2,44**
Hopscotch	LANG	2	3	1	3	2	2	3	3	3	**2,44**
Alice	LANG	2	3	1	3	2	2	3	3	3	**2,44**
Snap!	LANG	2	3	1	3	2	2	3	3	3	**2,44**
MakeWorld	CREA	2	3	2	3	2	2	2	2	3	**2,33**
Kodu Game Lab	LANG	2	3	2	3	0	2	3	3	2	**2,22**
Code.org - Courses	CHAL	2	2	2	3	1	2	2	3	2	**2,11**
ScratchJr	LANG	2	3	1	3	0	1	2	3	2	**1,89**
The Foos	CHAL	2	2	2	3	1	2	1	2	1	**1,78**
Blockly Games	CHAL	1	1	1	3	1	2	2	2	2	**1,67**
Made with Code	CHAL	2	1	1	2	1	2	2	2	2	**1,67**
Tynker - Activities	CHAL	2	2	2	3	1	1	1	2	1	**1,67**
Kodable	CHAL	1	1	2	3	0	2	2	2	1	**1,56**
Beetle Blocks	LANG	1	0	0	3	1	2	2	2	1	**1,33**
Waterbear	LANG	1	0	0	3	0	1	2	2	3	**1,33**
App Inventor	LANG	1	0	0	3	0	2	3	2	0	**1,22**
Cargo-Bot	CHAL	1	1	1	2	1	1	1	2	0	**1,11**
Lightbot	CHAL	1	1	1	3	0	1	1	2	0	**1,11**
Robozzle	CREA	0	0	1	3	2	1	1	2	0	**1,11**
SpriteBox	CHAL	1	1	1	3	0	1	1	1	0	**1,00**
Bee-Bot	CHAL	1	1	1	2	0	1	0	2	0	**0,89**
Kodetu	CHAL	0	1	1	2	1	1	1	1	0	**0,89**
Daisy the Dinosaur	CHAL	1	1	1	1	0	1	1	1	0	**0,78**

Tabla C.5. Implicación expresada en función de las dimensiones de diversión (de 0-min- to 3-max- cada una): Sen=Sensación, Fan=Fantasía, Nar=Narrativa, Ret=Reto, Com=Compañerismo, Des=Descubrimiento, Exp=Expresión, Ent=Entrega, His=Historia (cuento).

Observamos que se confirma la relación lógica que hay entre los juegos más difundidos y los más atractivos (Minecraft, Code.org, Tynker, Scratch). Además son más atractivos en general los lenguajes que permiten un desarrollo creativo libre, que posibilitan un engagement de mayor duración que el de un juego de retos cuya atracción básicamente acaba cuando se finalizan los retos. Lógicamente, vemos en la parte más baja las herramientas que menos recorrido proponen y

algunos otros (como App Inventor) cuya complejidad y cercanía a la realidad de bajo nivel aleja más el esfuerzo invertido del atractivo del resultado conseguido, desde un punto de vista de juego.

C.4.5. Pensamiento computacional

Type	Game	Loo	Alt	Deb	Mod	Var	Exp	Blo#	Evn	Thr	Par	Rec	Mes	Gen	OO	3D	Txt	Langs	Out
CHAL	Bee-Bot							4											
	Blockly Games	X	X	X	X	X	X	136									X	Js	
	Cargo-Bot		X	X	X			6				X							
	Code.org	X	X	X	X	X	X	>200	X	X							X	Js	
	Daisy	X		X				9	X										
	Kodable	X	X	X	X			7											
	Kodetu			X				9						X			X	Js	
	Lightbot			X	X			7				X							
	Made w/Code	X	X	X			X	>200											
	SpriteBox	X		X				7									X	Js, Sw	
	The Foos	X	X	X			X	20	X	X									
	Tynker - Act	X	X	X			X	>200	X	X									
CREA	MakeWorld			X	X			17	X	X		X							
	Robozzle		X	X	X			10				X							
LANG	Alice	X	X	X	X	X	X	>200	X	X	X	X	X		X	X	X	Java	
	App Inventor	X	X		X	X	X	>200	X	X		X	X		X			Java	X
	Beetle Blocks	X	X	X	X	X	X	120	X	X						X			
	Code Studio	X	X	X	X	X	X	>200	X	X		X	X		X		X	Js	
	Hopscotch	X	X		X	X	X	86	X	X		X							
	Kodu						X	>200	X	X						X			
	Scratch	X	X		X	X	X	130	X	X		X	X						X
	ScratchJr	X		X				26	X	X			X						
	Snap!	X	X	X	X	X	X	150	X	X		X	X						X
	Tynker	X	X	X	X	X	X	>200	X	X		X	X				X	Js, Py	X
	Waterbear	X	X			X	X	>200	X				X		X				
GAME	Minecraft	X	X	X		X	X	150	X	X						X	X	Js	

Table C.6. Aspectos del PC: Loo=Bucles, Alt=Alternativas, Deb=Depuración visual en ejecución, Mod=Módulos (subprogramas), Var=Variables, Exp=Expresiones, Blo#=# de bloques disponibles (expresividad del lenguaje), Evn=Eventos, Thr=Multihilo, Par=Sec. paralela, Rec=Recursividad, Mes=Paso de mensajes, Gen=Generalización, OO=OO, 3D=3D, Txt=Lenguaje texto equivalente, Langs=Lenguajes, Out=Salida a mundo físico (conexión posible con robots, sensores, arduino, etc.).

Es un aspecto fundamental para nuestro análisis revisar cuáles de las características concretas que se trabajan en el PC se incluyen en las herramientas analizadas. En la tabla C.6 podemos ver los aspectos que cada herramienta incluye, o no, junto a algunos datos significativos como el número de bloques diferentes que pueden utilizarse para programar (calculado contando todos los bloques diferentes que el

sistema permite utilizar), o si se puede ver el código textual equivalente en paralelo al programa visual que se va elaborando.

No hemos incluido en la tabla las secuencias, que tienen todas las herramientas analizadas (no podemos imaginar una herramienta de programación visual que no tenga secuencias). Hemos visto también que los flujogramas, una herramienta visual muy utilizada en programación y en aprendizaje de la programación a nivel conceptual, no se utiliza en ninguna de estas herramientas. Por otra parte, la recursividad se utiliza de dos maneras distintas: en aquellas herramientas que no soportan bucles, se realiza la repetición (virtualmente infinita) utilizando llamadas recursivas (es el caso de Cargo-Bot, LightBot, MakeWorld y Robozzle).

Los lenguajes soportan en general más características y utilizan muchos más bloques básicos para permitir mayor expresividad de programación (todos tienen más de 100 construcciones, excepto Hopscotch que las limita por su orientación a tablet, y ScratchJr que está orientado a los usuarios más pequeños). Los sistemas de retos, por su parte, tienen mucha menos expresividad, excepto los tres más desarrollados que soportan un gran número de niveles y diversifican mucho las construcciones que se pueden utilizar en cada reto: Code.org, Blockly, Tynker y Made with Code.

Reseñable también que todos los lenguajes soportan eventos, lo cual habla de la importancia de la programación orientada a eventos en la informática actual y también muestra que el concepto de evento que provoca una acción tiene un entendimiento muy natural para los aprendices. La gran mayoría de los lenguajes permiten multihilo, aunque sea de una forma transparente al aprendiz, que probablemente no necesite comprender el concepto para, implícitamente, utilizarlo. Solo algunos de los lenguajes (pero ninguna herramienta de retos) permiten paso de mensajes, orientación a objetos, 3D y conexión con sistemas físicos. Solo una herramienta (Alice) incorpora la construcción explícita de secuencia paralela (lanzar en el mismo espacio temporal varios bloques en paralelo), y también solo una de retos (Kodetu) trabaja el concepto de la generalización (que un mismo programa resuelva dos problemas diferentes).

C.4.6. Aspectos de diseño

La última dimensión analizada han sido algunas consideraciones de diseño de las herramientas, desde el punto de vista del planteamiento de la interfaz, de la secuencialización de la interacción con el usuario y de las opciones disponibles para profesores e investigadores (ver tabla C.7).

Type	Game	Prog. Interfaz	Tut	Help	Free	Reg	Grp	Feed	Dash	Use	Res
CHAL	Bee-Bot	Iconos*			X						
	Blockly Games	Ver - bloques	X	X	X						

Type	Game	Prog. Interfaz	Tut	Help	Free	Reg	Grp	Feed	Dash	Use	Res
	Cargo-Bot	Hor - iconos	X	X	X						
	Code.org	Ver - bloques	X	X	X	X	X	X	3		
	Daisy	Ver - bloques									
	Kodable	Hor - iconos	X	X		X	X	X	3		
	Kodetu	Ver - bloques	X				X	X	2		X
	Lightbot	Hor - iconos	X	X							
	Made w/Code	Ver - bloques	X	X							
	SpriteBox	Ver - iconos	X	X							
	The Foos	Hor - bloques	X	X		opc	X	X	1		
	Tynker - Act	Ver - bloques	X	X		opc	X	X	3		
CREA	MakeWorld	Ver - iconos	X		X	X					
	Robozzle	Hor - iconos	X		X	opc				X	
LANG	Alice	Ver - bloques	X	X	X						
	App Inventor	Ver - bloques		X	X	X					
	Beetle Blocks	Ver - bloques	X	X	X	opc					
	Code Studio	Ver - bloques	X	X	X	X				X	
	Hopscotch	Ver - bloques	X	X	X	X					
	Kodu	Grafo - iconos		X	X						
	Scratch	Ver - bloques		X	X	opc		X		X	
	ScratchJr	Hor - bloques		X	X						
	Snap!	Ver - bloques		X	X	opc					
	Tynker	Ver - bloques	X	X	X	X	X	X	2		
	Waterbear	Ver - bloques			X						
GAME	Minecraft	Ver - bloques	X	X	X	X	X	X	2		

Table C.7. Consideraciones de diseño: Tut=Tutorial integrado, Help=Ayuda integrada, Free=Navegación libre, Reg=Registro obligatorio, Grp=Posible creación de grupos (profesorado), Feed=Feedback de desempeño de usuarios, Dash=Dashboard profesorado (0-no a 3-completo), Use=Acceso público a datos de usuarios, Res=Acceso público de investigación a datos de usuarios.

Además de los datos indicados en la tabla, hemos también revisado las herramientas buscando algún tipo de adaptabilidad y no hemos encontrado ninguna que lo haga. Es decir, el software se comporta siempre igual independientemente de las características del usuario (edad, género, nivel educativo, diversidad funcional…) o de su comportamiento (falle o acierte, sea mejor o peor el programa, el juego no cambia el nivel posterior propuesto o no enseña diferente información o tutoriales). Lo único que se acerca a esta adaptabilidad tiene que ver con la puntuación, que analizaremos en la siguiente tabla.

Revisando el tipo de interfaz que se propone para la metáfora de "código" (el panel al que se pueden arrastrar las piezas para desarrollar el programa), vemos que la opción más habitual es la vertical (69,2%) que representa la secuencia de arriba abajo, y bastante menos la horizontal (23,1%) que representa la secuencia de

izquierda a derecha (en unos pocos casos con adaptación local a los idiomas que escriben de derecha izquierda). Hay dos herramientas especiales que no encajan en estos dos esquemas: Kodu que propone un creativo interfaz de grafo en círculo donde las opciones se van realizando por niveles y en cada nivel se muestran con iconos las disponibles, aglutinados en círculo; y Bee-Bot, que no tiene espacio de código explícito: es cada aprendiz quien tiene que recordar de memoria la secuencia de programa que "carga" en la abeja (igual que pasa en el dispositivo físico Bee-Bot).

La preferencia del interfaz vertical son los bloques (solo MakeWorld y SpriteBox, orientados a las franjas de edad inferiores, proponen iconos en vertical), y la del horizontal los iconos (excepto The Foos y ScratchJr que desarrollan unos bloques gráficamente elaborados para representar las estructuras repetitivas montadas sobre los iconos que se repiten). La tendencia es a utilizar estructuras horizontales con edades inferiores y verticales con edades superiores. Los bloques suelen tener forma visual de puzle y coloreado para diferenciar los tipos de construcción de forma visual. En algunos casos como Alice, App Inventor o Waterbear los bloques representan conceptos muy cercanos al código textual de bajo nivel correspondiente.

En la tabla también se puede observar las herramientas suministradas para los profesores. Las que permiten gestionar grupos de estudiantes y suministran información al profesor tienen además a menudo un dashboard online en el que el profesor, vía web, puede consultar información sobre el comportamiento de su grupo. Es bastante completo en el caso de Code.org y Kodable (progreso por tema, lecciones completadas), y especialmente detallista en Tynker (añadiendo las habilidades trabajadas y el nivel).

El acceso público a datos no es nada habitual: muy pocas herramientas muestran datos generales de uso o acceder a la información para investigación.

Finalmente, en la tabla C.8 podemos ver información del feedback que se va dando al usuario según se avanza en el sistema. Solo consideramos los juegos de reto, al ser los que pueden guiar al aprendiz sobre un comportamiento esperado.

Type	Game	Pre-level	Error	Success	Progress	Información de progreso accesible
CHAL	Bee-Bot	X		X	X	Estrellas, puntos
	Blockly Games	X		X	X	Niveles pasados
	Cargo-Bot	X	X	X	X	Estrellas, niveles pasados
	Code.org	X	X	X	X	Niveles pasados, Longitud de código
	Daisy	X		X		
	Kodable	X	X	X	X	Niveles pasados, puntos
	Kodetu	X		X		
	Lightbot			X		
	Made w/Code	X	X	X		
	SpriteBox			X	X	Niveles pasados, puntos

Type	Game	Pre-level	Error	Success	Progress	Información de progreso accesible
	The Foos	X	X	X	X	Niveles pasados, estrellas
	Tynker - Act	X	X	X	X	Niveles, estrellas, premios, certifs., conceptos
CREA	MakeWorld			X		
	Robozzle			X	X	Niveles pasados, Longitud de solución

Table C.8. Feedback al usuario. Pre-level=Feedback antes de cada nivel, Error=Feedback tras niveles fallados, Success=Feedback tras niveles exitosos, Progress=Info. explícita de progreso en el juego.

C.5. Conclusiones

A lo largo de este capítulo, hemos visto que existe un número creciente de opciones para un aprendiz del PC. Un camino de aprendizaje interesante puede ser empezar con algunos de los juegos de retos unos pocos días y progresar desde ahí a un lenguaje de programación visual, que puede conllevar semanas o meses de actividad. La diversión está asegurada, y hay muchas opciones diferentes, además de una diversidad de temáticas que hacen uso de temas de películas o videojuegos conocidos para reforzar la experiencia.

Sin embargo, existen todavía limitaciones que pueden ser consideradas para la próxima generación de juegos de PC. Hay un uso excesivo de primitivas de acción que tienen que ver con movimiento o rotación ortogonal (influenciados quizás, como todos nosotros, por Logo y su importancia histórica) en lugar de otras opciones auditivas, rítmicas o visuales, que permitirían también desarrollar pensamiento algorítmico y explorar otras habilidades diferentes a la visión espacial. En este sentido, hacemos notar algunas de las más recientes actividades incorporadas por Code.org y Made with Code con elementos multimedia. Encontramos también predominancia de bloques: herramientas como Kodu con sus diagramas de flujo abren posibilidades para diseñar nuevas herramientas de PC que combinen ambas y otras expresiones visuales, más allá de la única abstracción visual de bloques anidados. Otra carencia extendida es que, en este campo reciente, es especialmente importante investigar cómo los usuarios se comportan y cómo los sistemas facilitan el aprendizaje, luego sería deseable que las herramientas digitales facilitaran el uso de información de acceso abierto para análisis de aprendizaje, para permitir la mejora y saltos cualitativos en el diseño de nuevos niveles y herramientas. Una limitación final importante es la falta de aprendizaje adaptativo: prácticamente todas las herramientas se comportan igual, independientemente de la edad, conocimiento previo o habilidad mostrada por el usuario. Es importante que las herramientas comiencen a utilizar actividad previa del usuario para adaptar y mejorar significativamente la experiencia educativa, algo que debería resultar especialmente viable en este tipo de sistemas completamente digitales.

Un elemento clave en aprendizaje es la evaluación. En los juegos (que proponen un reto cuantificable, medible y observable) una evaluación adecuada es más factible. De hecho, está siendo considerada en varias plataformas: tamaño de código (número de bloques) en Code.org, estrellas por objetivos de tiempo o nivel en The Foos y otros, etc. La evaluación debería mejorarse para incluir más indicadores clave en habilidades específicas del PC, como eficiencia de programa (número de errores, número de cambios de código, diferencia de código con la mejor propuesta, etc.). En los lenguajes de programación la evaluación es mucho más compleja. Del mismo modo que un profesor que enseña lenguaje de programación basado en texto (como Python o Java) tiene una compleja tarea para evaluar a sus estudiantes, incluso más ocurre para un profesor/a de primaria o secundaria que no tiene por qué ser experto en computación y no está buscando las mismas cosas. De este modo, aunque puede utilizarse la remezcla para reforzar las habilidades incluidas en el PC [13], las herramientas abiertas aún carecen de las posibilidades automáticas, o incluso de la viabilidad para el profesorado de llevar a cabo una monitorización de la experiencia de aprendizaje. Se necesitan nuevos mecanismos de análisis y evaluación para verificar que los estudiantes superan la solución a un problema, y estudiar cómo lo resuelven y cómo progresan para conseguirlo. Probablemente, subsistemas del tipo propuesto por Dr.Scratch [14] se irán incorporando para facilitar indicadores que enriquezcan el proceso tanto para aprendices como para educadores. También hay una importante necesidad de criterios comunes de reconocer las habilidades del PC, así como para desarrollarlas y evaluarlas. A este respecto, herramientas fácilmente digitalizables como el CTTest propuesto por Román-González [15] pueden ser un buen complemento para evaluación en procesos de educación de duración media, usando tanto juegos como lenguajes.

Observando solo la categoría de juegos basados en retos de programación, vemos la importancia de generar más implicación después de la superación de los retos, incorporando técnicas conocidas de gamificación como puntuación, clasificaciones, propuestas de mejora o la incorporación de niveles creativos en los que el reto no esté limitado por objetivos fácilmente cuantificables; una línea en la que las entidades con más recursos como Code.org y Tynker ya están trabajando. Otro elemento clave en estos sistemas es el escalonamiento de los contenidos. Además del esfuerzo obvio de diseño de niveles (una necesidad compartida por juegos y educación), guiado por la experiencia de los diseñadores, es un reto sistematizar el proceso de asegurar el desarrollo del aprendizaje y la progresión de la motivación, en busca del flujo que tantos juegos consiguen. Es también necesario investigar las pautas para la generación automática de niveles/retos, en la línea ya conocida de generación procedural de muchos videojuegos. Sería también importante cómo desplegar las introducciones y tutoriales para maximizar el objetivo de aprendizaje e identificar el tipo de pensamiento que está aplicando el

aprendiz para resolver cada reto en progresión, como se estudia usando Kodu en [16]. Otra carencia final de los juegos basados en retos es que pocos sistemas permiten a aprendices o profesores la creación de nuevos retos, limitando con ello la experiencia y cerrando y limitando prematuramente el ciclo de aprendizaje. Herramientas como Make World o Robozzle, que permiten no solo jugar sino también editar nuevos mundos, serán parte de la próxima generación de juegos que incrementen la personalización de los retos a través de modificación o contribución creativa, en un tipo característico de remezcla.

Al respecto de la categoría de lenguajes de programación, hay limitaciones intrínsecas en los entornos visuales basados en bloques con respecto a los lenguajes basados en texto. Esto es mucho más notorio en proyectos de programación grandes debido a las limitaciones de visibilidad de código, navegación de código, o falta de control en modificaciones de lenguaje fuente [7]. La conversión bidireccional entre lenguajes visuales y textuales está disponible en un número creciente de plataformas como Code.org App Lab. Esta característica permite a los aprendices elegir la vista más útil, dependiendo de la complejidad del proyecto. Reseñamos además que las plataformas orientadas a retos vienen incluyendo dashboards de profesorado (Code.org, Kodable, Tynker) que todavía son muy escasos o no existentes en la categoría de lenguajes de programación, reflejando la falta de herramientas de evaluación ya mencionada.

Finalmente, hemos incluido en nuestro análisis una pequeña categoría de videojuegos que permiten programación visual dentro de sus mecánicas. A este respecto, Minecraft es un ejemplo interesante para los diseñadores de juegos: los buenos videojuegos pueden también ser diseñados considerando mecánicas específicas de PC. Creemos que este es uno de los retos de años venideros en diseño de juegos, no solo para juegos de construcción sino también en aventuras gráficas, RPGs, FPS, y otros muchos tipos de videojuegos.

Detalles de autores

Andoni Eguíluz*, Pablo Garaizar y Mariluz Guenaga
*Dirigir correspondencia a: andoni.eguiluz@deusto.es
Facultad de Ingeniería, Universidad de Deusto, Bilbao, España

Referencias

[1] Wing JM. Computational thinking. Communications of the ACM. 2006;49(2):33-35. DOI: 10.1145/1118178.1118215

[2] Buitrago Flórez F, Casallas R, Hernández M, Reyes A, Restrepo S, Danies G. Changing a generation's way of thinking: Teaching computational thinking through programming. Review of Educational Research. 2017. DOI: 10.3102/0034654317710096

[3] Wing JM, Stanzione D. Progress in computational thinking, and expanding the HPC community. Communications of the ACM. 2016;59(7):10-11. DOI: 10.1145/2933410

[4] Tedre M, Denning PJ. The long quest for computational thinking. In: Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16); New York, NY, USA. ACM; 2016. p. 120-129. DOI: 10.1145/2999541.2999542

[5] Denning PJ. Remaining trouble spots with computational thinking. Communications of the ACM. 2017;60(6):33-39. DOI: 10.1145/2998438

[6] Kazimogl C, Kiernan M, Bacon L, MacKinnon L. Learning programming at the computational thinking level via digital game-play. Procedia Computer Science. 2012;9:522-531. DOI: 10.1016/j.procs.2012.04.056

[7] Bau D, Gray J, Kelleher C, Sheldon J, Turbak F. Learnable programming: Blocks and beyond. Communications of the ACM. 2017;60(6):72-80. DOI: 10.1145/3015455

[8] Powers K, Gross P, Cooper S, Mcnally M, Goldman K, Proulx V, Carlisle M. Tools for teaching introductory programming: What works? ACM SIGCSE Bulletin. 2006;38(1):560. DOI: 10.1145/1124706.1121514

[9] García-Peñalvo FJ, Hughes J, Rees A, Jormanainen I, Toivonen T, Reimann D, Tuul M, Virnes M. Evaluation of existing resources (study/analysis). Belgium: TACCLE3 Consortium. 2016; DOI: 10.5281/zenodo.163112

[10] Sandoval-Reyes S, Galicia-Galicia P, Gutierrez-Sanchez I. Visual learning environments for computer programming. In: Electronics, Robotics and Automotive Mechanics Conference (CERMA); IEEE; 2011. p. 439-444. DOI: 10.1109/CERMA.2011.76

[11] Daly T. Using introductory programming tools to teach programming concepts: A literature review. The Journal for Computing Teachers. 2009;Fall:1-6

[12] Panitz M, Sung K, Rosenberg R. Game programming in CS0: A scaffolded approach. Journal of Computing Sciences in Colleges. 2010;26(1):126-132

[13] Dasgupta S, Hale W, Monroy-Hernández A, Hill BM. Remixing as a pathway to computational thinking. In: Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing (CSCW '16); New York, NY, USA: ACM; 2016. p. 1438-1449. DOI: 10.1145/2818048.2819984

[14] Moreno-León J, Robles G. Dr. Scratch: A Web Tool to Automatically Evaluate Scratch Projects. In: Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15); New York, NY, USA. ACM; 2015. p. 132-133. DOI: 10.1145/2818314.2818338

[15] Román-González M, Pérez-González JC, JiménezFernandez C. Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. Computers in Human Behavior. 2016:1-14

[16] Touretzky DS, Gardner-McCune C, Aggarwal A. Semantic reasoning in young programmers. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17); New York, NY, USA: ACM; 2017. p. 585-590. DOI: 10.1145/3017680.3017787

El amor no es la fuerza más grande que existe. El orden está por encima del amor. El amor necesita de un orden.

Joan Garriga

It is better to beg forgiveness, than ask permission

Grace Hopper

Apéndice

D

Metodología de revisión de la literatura

Para iniciar la investigación y para elaborar el estado del arte de esta investigación, realizamos una búsqueda extensiva de artículos en revistas y congresos. Para esa búsqueda se ha utilizado el buscador bibliográfico de la Biblioteca de la Universidad de Deusto, buscadores específicos de la ACM (*Association for Computing Machinery Digital Library*) e IEEE (*Institute of Electrical and Electronics Engineers*), así como buscadores especializados en artículos científicos: Google Scholar y ResearchGate.

Para realizar las búsquedas, utilizamos una serie de palabras clave que fuimos completando en un proceso iterativo hasta utilizar las siguientes: *"computational thinking", "programming learning", "programming teaching", "novice programmer", "programming in school", "programming in higher education", "coding for children", "computer science education", "Scratch", "Alice"*.

Con cada una de las referencias encontradas realizamos una rápida revisión del resumen (y en su caso, de las conclusiones). Tras esa revisión, descartamos la referencia o la incluimos en un listado en el que hemos recogido año, título, y reseña bibliográfica completa, además de un primer indicador, subjetivo, de interés aproximado de 0 a 5. Hemos mantenido ordenado el listado por año y título para evitar referencias duplicadas.

Tras elaborar el listado de referencias, realizamos un proceso de búsqueda de los artículos en formato electrónico, utilizando las herramientas de nuestra biblioteca digital universitaria y también referencias directas de Google Scholar y ResearchGate. Solicitamos los artículos no encontrados vía préstamo bibliotecario.

Tras la agrupación de la versión digital de los artículos encontrados, desarrollamos un script en Java que partiendo del texto plano de los artículos revisara las referencias bibliográficas de los mismos y las agrupara y ordenara, para realizar posteriormente un trabajo manual de revisión de estas referencias de cara a añadir las posiblemente interesantes al conjunto de ítems ya existente.

Con la lista final ya elaborada, desarrollamos un segundo script en Java que recorriera una por una todas las referencias obteniendo su enlace de Google Scholar y extrayendo el número de citas, añadiendo ese número a los datos disponibles de cada uno de los elementos.

Este proceso completo realizó en dos fases. La primera en el primer semestre de 2017, donde se obtuvo una primera serie de 653 referencias fechadas entre 1980 y 2017, y la segunda en el segundo semestre de 2019 donde completamos las 1.202 referencias manejadas (incluyendo nuevos artículos hasta unos pocos de 2020).

En la primera fase realizamos una revisión pormenorizada de las referencias con más citas, de donde extractamos ideas clave para el planteamiento de la investigación, a los elementos relevantes del PC que tener en cuenta y a la estructura de los aparatos estadísticos que utilizar. Se tomaron anotaciones de todos los artículos usados para su inclusión final en este documento de tesis y se revisaron las citas de estos artículos, incorporando algunas (nuevas o previamente desechadas) al proceso.

En la segunda fase, se realizó un paso adicional de identificar a los autores más relevantes en nuestro conjunto de referencias (asignando una ponderación por posición del autor en la lista de autores del artículo y sumando todas las apariciones), y desarrollamos una búsqueda específica de artículos de esos autores, ordenados por fechas. Desde esos listados revisamos e incluimos algunas referencias adicionales. Por último y ya con el listado definitivo, realizamos una revisión total de las referencias en orden de citado ponderado con fecha de publicación, revisando en primer término las menos relevantes, y haciendo una lectura exhaustiva de las más significativas, para su incorporación final en la investigación, completando algunos de los aspectos de este trabajo, y resultando por último en la composición de la bibliografía que se incluye al final de este documento.

Por otra parte y en paralelo, realizamos un trabajo de identificación de herramientas digitales de PC disponibles. Se realizó también un trabajo de búsqueda similar, pero en este caso con buscadores generales (Google, Yahoo y Bing), y con otro conjunto de palabras clave: “computational thinking tool”, “computational thinking platform”, “programming learning”, “online programming

tool", "code for novices", "digital programming tool", "coding in school", "coding in university", "tools for coding", "free tools for coding". También se añadieron a este listado herramientas mencionadas explícitamente en los artículos utilizados.

El procesado de esas herramientas se realizó en una sola fase, en el tercer trimestre de 2017. Para ello, nos registramos en cada una de las plataformas, realizamos las instalaciones y configuraciones correspondientes y dedicamos un tiempo a experimentar con cada una de ellas (unas pocas horas en cada una de las herramientas más sencillas y varios días en las más completas). Durante esa experimentación, tomamos las anotaciones pertinentes para elaborar las tablas y análisis expresados en el capítulo 2 de este trabajo y en el apéndice C.

Indicamos en la tabla D.1. los 26 autores más relevantes de nuestra bibliografía final (hemos extractado solo los que tienen al menos 3 artículos citados).

Autor	Puntuación	Nº Artículos	Nº prim. aut.
Grover, S.	37	8	7
Wilensky, U.	30	8	1
Repenning, A.	28	8	3
Wing, J.M.	23	5	4
Koh, K.H.	22	5	3
Weintrop, D.	20	4	4
Werner, L.	20	5	3
Cooper, S.	19	7	1
Denner, J.	18	5	1
Bau, D.	15	3	3
Denning, P.J.	14	3	2
Kölling, M.	14	3	2
Meerbaum-Salant, O.	14	3	2
Guenaga, M.	13	4	1
Armoni, M.	13	3	1
Basu, S.	13	3	1
Dagiene, V.	13	3	1
Guzdial, M.	13	3	1
Papert, S.	13	3	1
García-Peñalvo, F.J.	12	3	2
Bienkowski, M.	12	3	1
Dorling, M.	12	3	1
Selby, C.	11	3	2
Resnick, M.	11	3	1
Garaizar, P.	11	4	0
Basawapatna, A.	11	3	0

Tabla D.1. Autores más relevantes para este estudio, en función de su autoría y apariciones.

Cada uno da lo que recibe
Y luego recibe lo que da
Nada es más simple
No hay otra norma
Nada se pierde
Todo se transforma

Jorge Drexler. "Todo se transforma"

Apéndice

E

Definiciones y competencias del PC

E.1. Definiciones del PC

Mostramos en la tabla E.1 las definiciones del PC que hemos encontrado en la literatura, con el texto original en inglés de cada una de ellas, ordenadas por fecha de artículo.

Fuente	Definición del PC (en inglés)
(Denning, 2017)	The habits of mind developed from designing programs, software packages, and computations performed by machines
(ISTE, 2016)	Students develop and employ strategies for understanding and solving problems in ways that leverage the power of technological methods to develop and test solutions
(Riley y Hunt, 2014)	The way that computer scientists think, the manner in which they reason
(Mannila y cols., 2014)	A term covering a set of concepts and thinking processes from computer science that help in formulating problems and their solutions in different disciplines
(Sysło and Kwiatkowska, 2013)	A set of thinking skills that may not result in computer programming (…) should focus on the principles of computing rather than on computer programming skills

Fuente	Definición del PC (en inglés)
(Royal Society, 2012)	The process of recognising aspects of computation in the world that surrounds us, and applying tools and techniques from Computer Science to understand and reason about both natural and artificial systems and processes.
(Brennan y Resnick, 2012)	Involves three key dimensions: computational concepts (the concepts designers employ as they program), computational practices (the practices designers develop as they program), and computational perspectives (the perspectives designers form about the world around them and about themselves)
(Aho, 2012) y (Aho, 2011)	The thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms
(CSTA y ISTE, 2011) y (CSTA, 2011)	A problem-solving process that includes (but is not limited to) the following characteristics: *
(Wing, 2011)	Computational thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent.
(Perkovic y cols., 2010)	Intellectual skills necessary to apply computational techniques or computer applications to problems and projects in any discipline.
(Wing, 2006)	A way of solving problems, designing systems and understanding human behaviour by drawing on the concepts of computer science.

Tabla E.1. Definiciones del PC. *CSTA y ISTE definen el PC en función de sus características, como describiremos en la siguiente tabla.

E.2. Competencias del PC

Hemos encontrado en la bibliografía una serie de referencias a las competencias que incluye el PC. Podemos observar en la tabla E.2 un resumen de las más significativas que aparecen, organizadas en función de sus tipologías.

Fuente	Coms.	Abs	Dec	Alg	Eva	Gen	Aut	Par	Log	Eff	Dat	Sim	Pro	Otr
(Palts y Pedaste, 2020)	Meta-análisis	X	X	X	X	X							X	
(Fessakis y cols., 2018)	Anal. recursos	X	X	X	X	X	X		X		X	X	X	X
(Kalelioglu, Gulbahar y Kukul, 2016)	Meta-análisis	X	X	X	X	X	X			X		X	X	X
(Weintrop y cols., 2016)											X	X	X	X
(ISTE, 2016)			X	X			X				X			
(Csizmadia y cols., 2015)		X	X	X	X	X								
(Moreno-León y Robles, 2015)		X		X				X	X		X			X

Fuente	Coms.	Abs	Dec	Alg	Eva	Gen	Aut	Par	Log	Eff	Dat	Sim	Pro	Otr
(Gouws, Bradshaw y Wentworth, 2013)		X		X	X				X					X
(Grover y Pea, 2013)		X	X	X	X		X	X	X	X				X
(Selby y Woollard, 2013)		X	X	X	X	X								
(Barr y Stephenson, 2011)		X	X	X			X	X			X	X		
(CSTA y ISTE, 2011)				X		X	X			X	X	X		
(CSTA, 2011)				X		X				X	X	X	X	
(Wing, 2011)		X												
https://csunplugged.org/es/computational-thinking/		X	X	X	X	X			X					
http://www.cs.cmu.edu/~CompThink/		X		X										X

Tabla E.2. Competencias del PC encontradas en la literatura. Abs=Abstracción, Dec=Descomposición, Alg=P.Algorítmico, Eva=Evaluación, Gen=Generalización, Aut=Automatización, Par=Paralelización, Log=Lógica, Eff=Eficiencia, Dat=Datos, Sim=Simulación, Pro=Res. de problemas, Otr=Otros.

No siempre la referencia es exacta a cada competencia, incluimos a continuación un listado de los términos exactos indicados por cada autor, en el orden de la tabla:

- (Palts y Pedaste, 2020) *Abstraction. Decomposition. Algorithmic design, parallelization and iteration, automation. Testing and evaluation. Generalization. Problem formulation & Problem reformulation.*
- (Fessakis y cols., 2018) *Abstraction (47%). Problem decomposition (50%). Algorithmic thinking (66%), Pattern matching (21%). Evaluation (17%). Generalization (41%). Automation (12%). Logical reasoning (43%). Data representation (45%), Data analysis (14%), Data collection (3%). Modeling (33%), Simulation (22%). Problem translation (14%). Representation (38%), Testing (10%), Understanding people (7%), Sequencing (5%).*
- (Kalelioglu, Gulbahar y Kukul, 2016) *Abstraction (20%). Decomposition (6%). Algorithmic Thinking (14%) & Pattern Recognition (9%). Test & Debugging (4%). Generalization (4%). Automation (5%). Implementing Solutions (2%). Modeling (2%). Problem solving (13%). Design-based Thinking (7%), Conceptualising (6%), Analysis (5%), Mathematical Reasoning (4%).*
- (Weintrop y cols., 2016) *Data practices. Model & simulation practices. Computational problem solving practices. Systems thinking practices.*
- (ISTE, 2016) *Students break problems into component parts, extract key information, and develop descriptive models to understand complex systems or facilitate problem-solving. Students formulate problem definitions suited for technology-assisted methods such as data analysis, abstract models and algorithmic thinking in exploring and finding solutions. Students understand how automation works and use algorithmic thinking to develop a sequence of steps to create and test automated solutions. Students collect data or identify relevant data sets, use digital tools to analyze them, and represent data in various ways to facilitate problem-solving and decision-making.*

- (Csizmadia y cols., 2015) *abstractions, choosing good representations. decomposition. algorithmically. evaluation. generalizations, making use of patterns.*
- (Moreno-León y Robles, 2015) *Abstraction. Flow control. Paralellism. Logic. Data representation. User Interactivity, Synchronization.*
- (Gouws, Bradshaw y Wentworth, 2013) *Models and Abstractions. Patterns and Algorithms. Evaluations and Improvements. Inference and Logic. Tools and Resources, Process.*
- (Grover y Pea, 2013) *Abstractions and pattern generalizations (including models and simulations). Structured problem decomposition (modularizing). Algorithmic notions of flow of control. Debugging and systematic error detection. Systematic processing of information. Iterative, recursive, and parallel thinking. Conditional logic. Efficiency and performance constraints. Recursive thinking, Symbol systems and representations.*
- (Selby y Woollard, 2013) *Abstraction. Decomposition. Algorithmic thinking. Evaluation. Generalization.*
- (Barr y Stephenson, 2011) *Abstraction. Problem decomposition. Algorithms & procedures. Automation. Parallelization. Data collection, analysis, representation. Simulation.*
- (CSTA y ISTE, 2011) *Formulating problems in a way that enables us to use a computer and other tools to help solve them. Generalizing and transferring this problem-solving process to a wide variety of problems. Automating solutions through algorithmic thinking (a series of ordered steps). Identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources. Logically organizing and analyzing data. Representing data through abstractions such as models and simulations.*
- (CSTA, 2011) *Automating solutions through algorithmic thinking (a series of ordered steps). Generalising and transferring this problem solving process to a wide variety of problems. Identifying, analysing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources. Logically organising and analysing data. Representing data through abstractions such as models and simulations. Formulating problems in a way that enables us to use a computer and other tools to help solve them.*
- (Wing, 2011) *Abstraction.*
- https://csunplugged.org/es/computational-thinking/ *Abstraction. Decomposition. Algorithmic thinking. Evaluation. Generalization & Patterns. Logic.*
- http://www.cs.cmu.edu/~CompThink/ *Abstraction. Algorithmic thinking. Scale.*

Teniendo en cuenta el número de apariciones, mostramos en la tabla E.3 las competencias más relevantes, en ese orden.

Competencia del PC	Frecuencia
Pensamiento algorítmico	93%
Abstracción	80%
Descomposición	60%
Evaluación	53%
Generalización	53%
Datos	47%
Automatización, sistematización	40%
Simulación y modelos	40%
Lógica	33%
Resolución de problemas	33%
Eficiencia	27%
Paralelización	20%

Tabla E.3. Competencias esenciales del PC, ordenadas por aparición en la literatura.

Tu trabajo va a llenar gran parte de tu vida, y la única forma de estar realmente satisfecho con él es hacer lo que creas que es un gran trabajo. Y la única manera de hacer un trabajo genial es amar lo que haces. Si no lo has encontrado, sigue buscando. No te detengas. Al igual que con todos los asuntos del corazón, lo sabrás cuando lo encuentres. Y, como cualquier gran relación, sólo se pondrá mejor y mejor, conforme los años pasen. Así que sigue buscando hasta que lo encuentres.

Steve Jobs

Apéndice

F

Estudios con evaluación del PC

F.1. Estudios de evaluación del PC consultados

Reseñamos a continuación los datos de resumen de los 47 estudios con evaluación del PC referenciados en esta investigación. En la tabla F.1 se encuentran las referencias de cita de los estudios, junto a los datos de participantes, localización y duración de la experiencia de evaluación.

Nº	Fuente	NºPar.	Edad	Nivel educativo	País	Perfil adicional	Duración
1	(Alves y cols., 2020)	88.812				Apps públicas de App Inventor	
2	(Basu y cols., 2020)	14.787	10–12	prim. (2,3)	Hong Kong		
3	(Grover y Basu, 2017)	100	10–12	prim. (2,3)	EEUU		
4	(Touretzky, Gardner-McCune y Aggarwal, 2017)	39	8	prim. (2)	EEUU	-	4 ses.
5	(Román-González, Pérez-González y Jiménez-Fernandez, 2016)	1.251	10–16	prim. (3), sec. (1,2)	España		
6	(Fields y cols., 2016)	64	10–13	prim. (3), sec. (1)	EEUU	Camp de verano	1 sem.
7	(Dasgupta y cols., 2016)	173.053	4–90				
8	(Kalelioglu, 2015)	32	9	prim. (2)	Turquía	Extraacadémico	5 sem.
9	(Buffum y cols., 2015)	145	12–14	sec. (1)	EEUU		
10	(Spacco y cols., 2015)	1.386	>18	univ.	EEUU		15 sem.
11	(Mühling, Ruf y Hubwieser, 2015)	345	12–15	sec. (1,2)	Alemania		

Nº	Fuente	NºPar.	Edad	Nivel	País	Perfil adicional	Duración
12	(Armoni, Meerbaum-Salant y Ben-Ari, 2015)	120	15–16	sec. (2)	Israel	-	20 sem.
13	(Grover, 2015)	28	11-13	prim. (3)	EEUU	Clase opcional	20 sem.
14	(Grover, 2017)	28	11-13	prim. (3)	EEUU	Clase opcional	20 sem.
15	(Weintrop y Wilensky, 2015)	89		bach.			10 sem.
16	(Weintrop y Wilensky, 2015)	89		bach.			10 sem.
17	(Vihavainen, Luukkainen y Ihantola, 2014)	1166	12–76		Finlandia	MOOC de programación	15 sem.
18	(Grover, Cooper y Pea, 2014)	53	12–14	sec. (1)	EEUU	Clase opcional	6 sem.
19	(Bers y cols., 2014)	53	5–6	infantil	EEUU		
20	(Koh y cols., 2014)	39	>18	univ.	EEUU		15 sem.
21	(Miller y cols., 2014)	241	>18	univ.	EEUU	Cursos CS1	2 sem.
22	(Blikstein y cols., 2014)	346	>18	univ.	EEUU		10 sem.
23	(Koh y cols., 2014b)	134		prim. (3)	EEUU		4 sem.
24	(Zur-Bargury, Pârv y Lanzberg, 2013)	4082	12–13	sec. (1)	Israel		
25	(Franklin y cols., 2013)	22	12–14	sec. (1)	EEUU	Camp de verano. Perfil social infrarrepresentado	2 sem.
26	(Gujberova y Kalas, 2013)	128	8–10	prim. (2)	Eslovaquia		
27	(Boe y cols., 2013)	58			EEUU	Camp de verano	2 sem.
28	(Meerbaum-Salant, Armoni y Ben-Ari, 2013)	204	13–15	sec. (1,2)	Israel	-	20 sem.
29	(Werner, McDowell y Denner, 2013)	320	10–14	prim. (3), sec. (1)	EEUU	Clase opcional	
30	(Seiter y Foreman, 2013)	150	6-12	prim. 1-3			
31	(Fessakis y cols., 2013)	10	5–6	preesc.			1h. 15'
32	(Denner, Werner y Ortiz, 2012)	59	12–14	sec. (1)	EEUU	Chicas latinas, bajos ingresos, programa extraescolar volunt.	14 mes.
33	(Wilson, Hainey y Connolly, 2012)	60	8–11	prim. (1,2)	Escocia		8 sem.
34	(Dann y cols., 2012)	78	>18	univ.	EEUU		15 sem.
35	(Piech y cols., 2012)	370	>18	univ.	EEUU		5 mes.
36	(Brennan y Resnick, 2012)	31	8-17				
37	(Werner y cols., 2012)	311	10-14	prim. (3), sec. (1)	EEUU	Clases extraescolares	
38	(Burke, 2012)	10	12–14	sec. (1)		Chicos en programa extraescolar volunt.	7 sem.
39	(Meerbaum-Salant, Armoni y Ben-Ari, 2011)	46	14–15	sec. (2)	Israel	-	20 sem.
40	(Lewis, 2011)		11-12	prim. (3)		Curso de verano	3 sem.
41	(Marshall, 2011)	>500		sec.	EEUU		1 h.
42	(Panitz, Sung y Rosenberg, 2010)	14	>18	univ.	EEUU		
43	(Lewis, 2010)	50	10-12	prim. (3)	EEUU	Curso de verano	36 h.

Nº	Fuente	NºPar.	Edad	Nivel	País	Perfil adicional	Duración
44	(Garlick y Cankaya, 2010)	155	>18	univ.	EEUU		2 sem.
45	(Maloney y cols., 2008)	80	8–18		EEUU	Jóvenes afroam. en clubhouse, zona suburbios de LA	18 mes.
46	(Parsons y Haden, 2007)	72	>18	univ.	EEUU		12 sem.
47	(Moskal, Lurie y Cooper, 2004)	36	>18	univ.	EEUU		6 sem.

Tabla F.1. Datos de participantes y duración de los estudios consultados.

En la tabla F.2 encontramos la herramienta de PC utilizada (si la hay), la asignatura o temática del curso o actividad educativa en la que se enmarca y el tipo de investigación que se plantea. Utilizamos en esta tabla y sucesivas el número de estudio codificado en la primera tabla resumen.

Nº	Lenguaje / Plataforma	Asignatura / tema	Tipo investigación
1	App Inventor		Análisis cualitativo de artefactos
2			Validación cuestionario
3	Scratch	Programación	Validación cuestionario
4	Kodu Game Lab	Programación	Experimental
5			Validación cuestionario
6	Scratch	Programación	Estudio de caso
7	Scratch	Programación	Estudio de caso
8	code.org	Programación	Experimental
9		Programación	Validación cuestionario
10	CloudCoder	Programación	Estudio de caso
11			Validación cuestionario
12	Java o C#	Programación	Mixta
13	Scratch	Programación	Experimental
14	Scratch	Programación	Experimental
15	Snap! y Java	Programación	Estudio de caso
16	Snap! y Java	Programación	Validación cuestionario
17	Test My Code en Netbeans	Programación	Estudio de caso
18	Scratch	Programación	Investigación-acción
19	LEGO kit robótica y lenguaje CHERP	Programación y robótica	Estudio de caso
20	AgentSheets / AgentCubes	Diseño de juego educativo	Estudio de caso
21		Informática	Experimental
22	Java, Karel The Robot	Metodología de programación	Estudio de caso
23	AgentSheets / AgentCubes	Programación	Análisis cualitativo de artefactos
24		Programación	Estudio de caso
25	Scratch	Interdisciplinar	Estudio de caso
26		Programación	Estudio de caso
27	Hairball	Programación	Análisis cualitativo de artefactos

Nº	Lenguaje /	Asignatura /	Tipo investigación
28	Scratch	Programación	Mixta
29	Alice	Programación	Análisis cualitativo de artefactos
30	Scratch	Programación	Estudio de caso
31	Logo	Matemáticas	Estudio de caso
32	Stagecast Creator	Programación	Análisis cualitativo de artefactos
33	Scratch	Programación	Estudio de caso
34	Alice	CS1	Experimental
35	Java	CS1	Estudio de caso
36	Scratch / Scrape		Estudio de caso
36b	Scratch		Estudio de caso
36c	Scratch		Estudio de caso
37	Alice	Programación	Análisis cualitativo de artefactos
38	Scratch	Inglés	Estudio de caso
39	Scratch	Programación	Estudio de caso
40	Scratch, Logo, Snap	Programación	Experimental
41	AgentSheets	Programación	Estudio de caso
42	GameMaker y C#	CS1	Estudio de caso
43	Scratch y Logo	Programación	Experimental
44	Alice	CS1	Experimental
45	Scratch	Programación	Estudio de caso
46	Alice	CS1	Estudio de caso
47	Alice	CS1	Experimental

Tabla F.2. Herramienta, asignatura/temática y tipo de investigación de los estudios consultados.

Podemos ver la descripción de la intervención en la tabla F.3, junto a la identificación de la dimensión del PC evaluada, de acuerdo a la diferenciación de tres dimensiones (conceptos, prácticas, perspectivas) propuesta por Brennan y Resnick (2012).

Nº	Descripción intervención	PC		
		Concep.	Práct.	Persp.
1	Validación de una herramienta de rúbrica del PC en proyectos de App Inventor	X		
2	Validación cuestionario	X		
3	Validación cuestionario	X		
4	Exp: Creación juego con Kodu con reglas explícitas Cont: Creación juego con Kodu sin reglas explícitas	X		
5	Validación cuestionario	X	X	
6	Retos de programación abiertos con Scratch	X		
7	Observación de repositorios de Scratch que usan remezclado	X		
8	Uso de las actividades secuencializadas de code.org para un curso de 5 semanas	X		
9	Cuestionario de conceptos de programación usando bloques visuales	X		
10	Análisis de los ejercicios de programación editados y entregados por los estudiantes		X	
11	Validación cuestionario	X		

Nº	Descripción intervención	PC		
12	Exp: Programación con formación previa en Scratch Cont: Programación sin formación previa	X		
13	Exp: Formación en Scratch con múltiples herramientas de evaluación (multifaceta): encuestas de percepción, cuestionarios del PC, rúbrica de ejercicios y proyectos abiertos. También un cuestionario "Preparation of Future Learning" (PFL) para evaluar transferencia del PC a un lenguaje textual	X	X	
15	Comparación de programación basada en bloques y textual		X	
16	Cuestionario de conceptos de programación usando a veces bloques visuales y a veces lenguaje de texto	X		
17	Seguimiento de ejercicios de los participantes usando log de grano fino	X		
18	Primer curso: Solo algunos vídeos online Segundo curso: Materiales online y vídeos cortos	X		
19	Clase guiada por profesorado y asistentes especializados	X		
20	Medición automática de patrones de código (CTP) en clases de programación de juegos para uso educativo	X		
21	Incorporación ejercicios de pensamiento creativo asociado al PC	X		
22	Captura de snapshots de código progresivo en ejercicios propuestos de programación Java y Karel The Robot		X	
23	Entorno de seguimiento de desempeño de estudiantes en tiempo real para profesorado	X		
24	Examen del PC a nivel nacional en Israel	X		
25	Desarrollar PC en un contexto no informático	X		
26	Concurso de PC a nivel nacional en Eslovaquia (test Bebras). Prueba para diseño del test con graduación progresiva	X	X	
27	Evaluación herramienta (Hairball) para análisis estático de proyectos de Scratch	X		
28	Programación con Scratch	X		
29	Medición automática de logs en actividades con Alice		X	
30	Evaluación de proyectos Scratch contra un modelo propio de progresión del PC	X		
31	Clase guiada por profesorado con pizarra interactiva		X	
32	Creación de juegos	X		
33	Sesiones de Scratch con juegos de ejemplo y se les propone que desarrollen sus propios juego, partiendo de los existentes. Trabajan en pares	X		
34	Exp: Alice con Java Cont: Solo Java	X		
35	Análisis de registros de código de dos ejercicios de programación en cursos de introducción a la informática		X	
36	Scrape para análisis de portfolios	X		
36b	Entrevistas personales basadas en artefactos (elementos concretos de algunos de los proyectos)		X	
36c	Escenarios de diseño partiendo de proyectos predefinidos		X	
37	Estudiantes realizan el "Fairy Assessment", una serie de tareas sobre programas prediseñados en Alice	X		
38	Cuenta cuentos digital	X		X
39	Programación con Scratch		X	
40	Exp: Con pair programming Cont: Sin pair programming	X		

Nº	Descripción intervención	PC		
41	Reconocimiento de patrones del PC en clips de vídeo, tras programar un juego con AgentSheets	X		
42	Escalonar el aprendizaje con GameMaker y programación de juego en lugar de solo C#	X		
43	Exp1: Primero Scratch y luego Logo (solo se evalúa Scratch) Exp2: Primero Logo y luego Scratch (solo se evalúa Logo)	X		
44	Exp1: Introducción con Alice Exp2: Introducción con pseudocódigo (tradicional)	X		
45	Programación libre con Scratch	X		
46	Pruebas de Alice dentro de un curso de programación con Pascal	X		
47	Exp: Con curso de Alice en paralelo a CS1 Cont1: Sin curso de Alice, con riesgo académico Cont2: Sin curso de Alice, sin riesgo académico	X		

Tabla F.3. Descripción de la intervención y dimensión del PC que trabaja: concepto, práctica o perspectiva, según define (Brennan y Resnick, 2012).

En la tabla F.4 indicamos el tipo y descripción de la evaluación y los datos de analítica utilizados.

Nº	Evaluación			
	S/N	Tipo	Datos Analítica	Descripción
1	sí		proyectos App Inventor	Rúbrica sobre elementos incluidos en los proyectos
2		Formulario resp. cerrada		Sin relación con herramientas de PC particulares
3	sí	Formulario resp. cerrada		Específico con bloques de Scratch y cols. elementos particulares
4	sí	Formulario resp. cerrada		Específico con reglas de Kodu
5	sí	Formulario resp. cerrada		Específico con bloques estilo code.org
6	sí		snapshots de proyecto de Scratch en cada cambio	
7			Proyectos de Scratch compartidos	
8		RTSSTPS - Reflective Thinking Skill Scale Towards Problem Solving		
9	sí	Formulario resp. cerrada		Específico con bloques visuales
10	sí		Código de los estudiantes y evolución de grano fino	
11	sí	Formulario resp. cerrada		Específico con pseudocódigo
12	sí	Tres formularios pre-in-posttest		Preguntas de distintos tipos sobre conceptos de programación Java o C#

Nº	Evaluación			
	S/N	Tipo	Datos Analítica	Descripción
13	sí	Múltiple (cuestionarios, rúbricas, abierta)		
14	sí	Ídem 13		
15	no			
16	sí	Formulario resp. cerrada		Específico con pseudocódigo y bloques visuales (basado en errores habituales)
17	no		Logs de Java de grano fino	
18	sí	Formularios resp. cerrada pre-post		Preguntas de distintos tipos sobre scripts de Scratch
19	sí	Evaluación personalizada con rúbrica Likert		Revisión de código tangible de cada sesión y participante
20	sí		Juegos respuesta final a actividades	
21	sí	Formulario resp. cerrada		Preguntas conceptuales y de resolución de problemas de PC
22	no		Evolución de código entre estados consecutivos	
23	sí		Medición de grafo CTPA con herramienta específica (REACT)	
24	sí	Formulario resp. cerrada		Preguntas de distintos tipos sobre scripts de Scratch
25	sí	Evaluación personalizada con rúbrica Likert		
26	sí	Formulario resp. cerrada	Logs de tiempos e interacciones de usuarios con el test	Preguntas de distintos tipos sobre conceptos de programación
27	sí		Proyectos finales Scratch	
28	sí	Tres formularios pre-in-posttest		Preguntas de distintos tipos sobre scripts de Scratch
29	sí		Registro de ejecución de bajo nivel	
30	sí	Revisión manual de código		Rúbrica con evidencias en base a los bloques utilizados
31	no			
32	sí	Revisión manual de código		
33	sí	Rúbrica en base a código Scratch		Incluye elementos del PC (secuencia, iteración, variables…) y valoraciones
34	sí	Examen convencional Java		
35	sí	Análisis de código	Registro de repositorios en el desarrollo de práctica de programación	ML con modelo de Markov
36	sí	Análisis de código en portafolio	Bloques utilizados y frecuencia	

Nº	Evaluación			
	S/N	Tipo	Datos Analítica	Descripción
36b	sí	Entrevistas personales		Preguntas sobre Scratch y sobre proyectos personales
36c	sí	Entrevistas personales		Selección de escenario y modificación de proyectos previos modelo
37	sí	Revisión manual de código		Rúbrica en base a evaluación manual
38	no			
39	sí	Revisión manual de código		
40	sí	Formularios resp. cerrada		Formularios diarios según avanza el curso
41	sí	Formulario resp. Abierta		Preguntas de reconocimiento de patrones en clips vídeo
42	sí	Formularios resp. cerrada		
43	sí	Formularios resp. Numérica		Preguntas de comprensión de funcionamiento de código (en Scratch o en Logo)
44	sí	Desarrollo de algoritmo		Se plantea un problema que se puede realizar de la forma que el estudiante desee
45	sí		ficheros de resumen de proyecto de Scratch (por semana)	
46	sí	Observación de práctica entregada		Se observa el problema resuelto en Alice valorando uso de ifs y loops
47	sí	Evaluación convencional de CS1		Se miden resultados finales de CS1 y se comparan los que han hecho o no Alice

Tabla F.4. Datos de evaluación y descripción de analítica de aprendizaje, si existe.

Por último, en la tabla F.5 vemos la indicación de qué se mide al respecto del PC y un resumen de los resultados del estudio.

Nº	Qué se mide	Resultados
1	Criterios de algoritmia, programación y apps móviles	La herramienta tiene suficiente validez y fiabilidad para medir los conceptos básicos del PC partiendo solo del código de proyectos
2	Conocimientos, competencias y habilidades en 4 áreas: remezcla, pensamiento algorítmico, abstracción y modularización y pruebas y depuración	
3	Equívocos en bucles, variables y lógica booleana	A pesar de los bloques las dificultades semánticas (entender variables, bucles y lógica) persisten
4	Comprensión, orden de ejecución, resolución de conflictos, secuencias anómalas	Reglas sencillas permiten trabajar en primaria (3º). Sugerencias sobre secuenciación de ejercicios

Nº	Qué se mide	Resultados
5	Aprendizaje de conceptos y algunas prácticas del PC	
6	Corrección y errores en tres conceptos: inicialización, eventos, paralelismo	Observaciones particulares sobre errores y estilos de programación
7	Diferenciación de seis ejes relacionados con el PC en función de los bloques de Scratch utilizados: bucles, paralelismo, eventos, condicionales, operadores y datos	La remezcla influye positivamente en el aprendizaje de las habilidades incluidas en el PC
8	Influencia del PC en el pensamiento reflexivo sobre solución de problemas	No hay diferencia significativa de pensamiento reflexivo sobre solución de problemas tras hacer el curso
9	Conceptos elementales del PC	Se elaboran un protocolo de 7 pasos para diseñar, refinar, validar instrumentos de evaluación del PC.
10	Análisis de evolución del código	Posibilidades del estudio de grano fino de cambios en código textual
11	Aprendizaje de conceptos del PC	
12	variables, conditionales, repetitivas con contador y repetitivas condicionales	No hay mejora general para los que tenían form. previa, excepto aspectos puntuales como bucles, y mejor resultado en creación relacional
13	Comprensión del PC en múltiples facetas en base a capacidad de programación con Scratch	La evaluación multifaceta suministra una información mucho más profunda. La autora propone usar "sistemas de evaluación"
14		
15	Diferencias de aprendizaje con programación visual basada en bloques o lenguaje de programación	los bloques son más fáciles de leer, sus formas son útiles, son más fáciles de componer, y evitan tener que memorizar. También tienen inconvenientes: la programación es menos potente, más lenta y se siente menos auténtica.
16	Comprensión, lógica condicional, procedimientos, lógica iterativa y variables	Rendimiento y comprensión mejor en preguntas con bloques visuales que en las de texto
17		
18	Conceptos de programación	La mejora de la metodología mejora los resultados
19	Resultados y observación personalizada de depuración, correspondencia, secuenciado y flujo de control	La programación tangible puede ser introducida con éxito en etapas prelectoras
20	CTPA (análisis de patrones de CT)	Alta correlación entre evaluación humana y automática con CTPA
21	Resultado en los tests y en las evaluaciones estándar del curso universitario	Los ejercicios de pensamiento creativo pueden mejorar la comprensión del PC en disciplinas no informáticas
22	Cambios de código, comparado con evaluación (manual)	Al examinar el proceso de programación en lugar del resultado final, se encuentran datos contraintuitivos sobre el comportamiento de los estudiantes de programación. Pueden encontrarse patrones con mejor predicción de competencia que los exámenes convencionales
23	CTPA (análisis de patrones de CT)	Herramienta que ayuda sustancialmente al seguimiento del profesorado mientras se desarrollan las experiencias de programación

Nº	Qué se mide	Resultados
24	Conceptos de programación: secuencias, alternativas, repetitivas	Identificación de mejoras en prueba y en retos de los educadores para años posteriores
25	4 áreas de evaluación: eventos, inicialización, sincronización y animación	Se puede incluir y evaluar el PC en aprendizajes multidisciplinares fuera del ambiente escolar
26	Tiempos, interacciones, intentos al resolver cuestionario de conceptos de programación	Es importante analizar los procesos cognitivos de los estudiantes en las primeras fases de aprendizaje de programación
27	Conceptos del proyecto Scratch: inicialización, sincronización, emisión y animación	La herramienta es muy útil para detectar implementaciones correctas, pero debe mejorarse y completarse con evaluación manual
28	Conceptos de programación: inicialización, repetición, condicionales, comunicación, eventos, concurrencia	El aprendizaje es eficaz para los conceptos medidos, en distintas profundidades
29	Acciones de alto nivel inferidas automáticamente desde los logs	Gran potencial del análisis de logs para entender cómo los jóvenes aprenden con la tecnología, especialmente en entornos de aprendizaje no estructurado
30	Complejidad del PC incorporado (modelo PECT)	Modelo de progreso del PC para todo el ciclo de primaria: la complejidad crece congruentemente con el curso
31		
32	Categorización específica con 24 ítems: paralelismo, aleatoriedad, variables globales, nombres de reglas…	La programación de juegos es una metodología prometedora para motivar a estudiantes de extracto social desfavorecido para introducirlos en PC. Aunque los aspectos más complejos de la programación no fueron abordados
33	Uso de los conceptos del PC	Los niños/as son capaces de adaptar juegos existentes y en edades finales de primaria, proponer juegos sencillos originales
34	Resultados en examen final Java	Ambos lenguajes mejoran rendimiento > 10%. Remarcable ya que podría asumirse que el tiempo que se invierte en Alice disminuye el tiempo dedicado a Java
35	Patrones de evolución de código por cambios entre estados	Los patrones que encuentran predicen mejor el desempeño de los aprendices en sus exámenes que las evaluaciones recibidas en las prácticas de programación entregadas
36	Bloques utilizados y frecuencia de uso	Planteamiento de marco de evaluación con ventajas e inconvenientes
37	Rúbrica de las tareas, que miden compresión, diseño y resolución de problemas	Se pueden diseñar retos progresivos para evaluar PC. Parece haber mejores resultados por pares que de forma individual
38		
39	Hábitos de programación	Programación bottom-up y programación de grano extremadamente fino
40	Comprensión de los conceptos del PC	Contrariamente a lo esperado, los resultados de la programación por pares son peores
41	Capacidad de reconocer patrones del PC	Relacionar patrones del PC con situaciones de la vida real puede afianzar el conocimiento del PC

Nº	Qué se mide	Resultados
42	Comprensión de los conceptos de programación	Entornos como GameMaker pueden significar un buen paso previo en el aprendizaje de un lenguaje de programación textual como C#
43	Comprensión de los conceptos de programación	La comprensión del PC puede mejorar en aprendices con lenguajes de bloques sobre lenguajes de texto
44	Comprensión de los conceptos de programación	Se comportan peor los estudiantes que se han introducido con Alice que los que se introducen con pseudocódigo
45	Diversidad de elementos de programación utilizados	Los jóvenes de entornos de alta complejidad pueden aprender y usar entornos abiertos como Scratch e ir incrementando la complejidad de sus actividades
46	Capacidad de incluir construcciones algorítmicas en lenguajes visuales	No hay una conexión profunda del PC entre lenguajes de texto y lenguajes visuales como Alice
47	Comprensión de programación CS1	Aprender Alice mejora los resultados de CS1 convencional, especialmente en estudiantes en riesgo (sin exp. previa de programación o con bajas calificaciones matemáticas)

Tabla F.5. Descripción de qué se evalúa y resumen de resultados del estudio.

La libertad vuela como las cometas. Vuela porque está atada. Usted coja una cometa y láncela, no vuela. Pero átela una cuerda y entonces resistirá al viento y subirá. Cuál es la cuerda de la cometa de la libertad: la igualdad y la fraternidad. Es decir, la libertad responsable frente a los demás.

José Luis Sampedro

Apéndice

G

Glosario y abreviaturas

G.1. Glosario

3D	3 dimensiones, tipo de juego y programación gráfica que simula el mundo tridimensional en las dos dimensiones de pantalla.
ACP	Técnica estadística de Análisis de Componentes Principales (PCA en inglés).
Alfabetización informática	Incorporación en el proceso educativo de habilidades de construcción tecnológica.
API	Conjunto de reglas de comunicación entre aplicaciones (*Application Programming Interface*).
App	Programa instalable, especialmente para el caso de dispositivos móviles (abreviatura de *Application*).
arcade	Videojuego recreativo característico de los años 1970-1980.
AST	Árbol abstracto de sintaxis (*Abstract Syntax Tree*)
blended analytics	Analítica combinada, tipo de investigación que combina analítica de datos con estudio dirigido por hipótesis.
bottom-up	Aproximación de diseño en la que primero se detallan las partes individuales, luego se enlazan para formar componentes más grandes y así sucesivamente.
Computer literacy	Alfabetización informática.
CTTest	Cuestionario de medición del PC propuesto por Román-

	González, Pérez-González y Jiménez-Fernández
dashboard	Cuadro de mando, aplicación digital que permite observar y a veces operar un proceso de forma sintética y visual.
DGBL	Aprendizaje basado en juego digital (*Digital Game Based Learning*)
EDM	Minería de datos educativos (*Educational Data Mining*)
emoticono	Icono que transmite expresión facial y/o emocional (neologismo derivado del inglés *emoticon*, contracción de emoción e icono).
engagement	Implicación. Usada en el contexto informático para describir la capacidad que tiene un programa o juego de conseguir que sus usuarios lo utilicen de una forma continua y entregada.
feedback	Retroalimentación, devolución de información al usuario.
framework	Entorno de trabajo. Conjunto de conceptos y/o herramientas software relacionadas y aplicables a un ámbito concreto.
GBL	Aprendizaje basado en juego (*Game Based Learning*).
IA	Inteligencia artificial
IDE	Entorno integrado de desarrollo (*Integrated Development Environment*)
Java	Lenguaje de programación (utilizado en KodetuWin).
JavaScript	Lenguaje de programación (utilizado en Kodetu/KG/KG2).
JSON	Formato de texto para intercambio de datos en JavaScript (*JavaScript Object Notation*).
Kodetu	Herramienta online desarrollada por nuestro equipo de investigación en 2014 basada en maze de blockly. Accesible en la web http://kodetu.org
KodetuGen	Herramienta online desarrollada por nuestro equipo de investigación como una segunda versión de Kodetu, que incorpora generalización. Accesible hasta 2017 en la web http://gen.kodetu.org
KodetuGen2	Herramienta online desarrollada por nuestro equipo de investigación como una versión mejorada de KodetuGen, que incorpora generalización y permite registro y selección

	de reto. Accesible desde 2018 en la web http://kodetu.org
KodetuWin	Herramienta de análisis desarrollada por nuestro equipo de investigación para el procesado y análisis previo de la información registrada en las distintas versiones de Kodetu.
LA	Analítica de aprendizaje (*Learning Analytics*).
Laberinto dual	Laberinto creado en KodetuGen y KodetuGen2, formado por dos laberintos diferentes que necesita para su solución un código único que resuelva ambos.
liveness	Característica de algunos entornos de programación que permite que una parte del sistema pueda ejecutarse o modificarse en tiempo real mientras otras partes del sistema se están a su vez ejecutando.
marca temporal	(En inglés *timestamp*) Identificación de tiempo concreto (fecha y hora) en el que un aprendiz realiza una acción, normalmente con precisión de milisegundos.
MMORPG	Juego de rol masivo multijugador en línea (*Massively Multiplayer Online Role-Playing Game*).
MOOC	Curso en línea masivo y abierto (*Massive Open Online Course*)
online	En línea, conectado a una red (generalmente, internet).
open source	Código abierto, modelo de software basado en la colaboración abierta.
PBL	Aprendizaje basado en proyectos (*Project Based Learning*).
PC	Pensamiento computacional.
PC desenchufado	Pensamiento computacional trabajado con materiales no digitales
PCA	*Principal Component Analysis*. Ver ACP.
PISA	Estudio mundial de la OCDE que mide anualmente el rendimiento académico de los estudiantes en matemáticas, ciencia y lectura (acrónimo de *Programme for International Student Assessment*).
plugin	Módulo informático que se integra con un software existente para añadirle o modificar su funcionalidad.
profiling	Análisis de rendimiento de software.
remezcla	Capacidad existente en algunas plataformas de partir de

	programas/soluciones previos para elaborar un programa/solución nuevo.
remix	Remezcla.
RTS	Tipo de juegos de estrategia en tiempo real (*Real Time Strategy*).
scaffolding	Andamiaje. Estructura temporal de contenidos que se define con el objetivo de mejorar el aprendizaje.
snapshot	Registro de información de un sistema en un momento preciso. Puede tratarse de una imagen (una captura de pantalla) o de un conjunto de datos.
sprite	Conjunto de imágenes que forman el aspecto visual de un personaje u objeto de un juego, animación o simulación. Puede ser estático (una sola imagen) o dinámico (incluyendo las imágenes correspondientes a una o varias de sus animaciones).
SQL	Lenguaje estandarizado de gestión de base de datos relacionales (*Structured Query Language*).
STEM	Áreas educativas relacionadas con la ciencia, tecnología, ingeniería y matemáticas *(Science, Technology, Engineering, & Mathematics).*
tipo Kodetu	Clasificación de niveles de Kodetu en función de la que diferenciamos la complejidad de los niveles de acuerdo a cuatro categorías A, B, C, D. El tipo A incluye los niveles que se resuelven solo con bloques secuenciales, el B añade la repetición, el C la alternativa simple y el D la alternativa múltiple.
unplugged	"desenchufada", referencia a PC desenchufado

G.2. Abreviaturas utilizadas en el documento

ad	Abreviatura utilizada en Kodetu para el bloque de movimiento hacia adelante (*forward*).
DT	Desviación típica.
gi	Abreviatura utilizada en Kodetu para el bloque de girar a la izquierda (*left*).
gd	Abreviatura utilizada en Kodetu para el bloque de girar a

	la derecha (*right*).
KG	KodetuGen.
KG2	KodetuGen2.
K-W	Prueba estadística no paramétrica de Kruskal–Wallis.
M	Media aritmética.
Me	Mediana estadística.
M-W-W	Prueba estadística no paramétrica de Mann–Whitney–Wilcoxon.
rep	Abreviatura utilizada en Kodetu para el bloque de repetir hasta que se encuentra la meta (bucle) (*repeat*).
si-ad	Abreviatura utilizada en Kodetu para el bloque de Condicional si hay camino adelante (*if path ahead*).
si-i	Abreviatura utilizada en Kodetu para el bloque de Condicional si hay camino a la izquierda (*if path left*).
si-d	Abreviatura utilizada en Kodetu para el bloque de Condicional si hay camino a la derecha (*if path right*).
si-no	Abreviatura utilizada en Kodetu para el bloque de Condicional de dos ramas (segunda parte) (*else*).

Vuelo.
A donde el corazón quiera llevarme,
contando con la mente que organiza,
jugando con los vientos;
fluyendo en dirección, pero sin prisa,
en sintonía con mi naturaleza,
a donde la tormenta me permita.

Andoni Eguíluz Morán, enero de 2015

Apéndice

H

Contribuciones científicas

H.1. Publicadas

Como resultado de la investigación expuesta en el capítulo 3 de esta tesis se ha publicado el siguiente artículo de revista (indexada WoS, Q e IF):

- Andoni Eguíluz, Mariluz Guenaga, Pablo Garaizar y Cristian Olivares-Rodríguez (2020). "Exploring the Progression of Early Programmers in a Set of Computational Thinking Challenges via Clickstream Analysis" in IEEE Transactions on Emerging Topics in Computing, vol. 8, no. 1, pp. 256-261, Jan.-March 2020. DOI: 10.1109/TETC.2017.2768550

Como hemos comentado en los capítulos 2 y apéndice C de este documento, el capítulo de libro:

- Andoni Eguíluz, Pablo Garaizar y Mariluz Guenaga (diciembre 2017). An Evaluation of Open Digital Gaming Platforms for Developing Computational Thinking Skills. En *Simulation and Gaming*, Dragan Cvetković, IntechOpen, DOI: 10.5772/intechopen.71339. Disponible en: https://www.intechopen.com/books/simulation-and-gaming/an-evaluation-of-open-digital-gaming-platforms-for-developing-computational-thinking-skills

En la línea de originalidad en el PC relacionada con la creatividad, el artículo de revista (indexada WoS, Q e IF):

- Arnon Hershkovitz, Raquel Sitman, Rotem Israel-Fishelson, Andoni Eguíluz, Pablo Garaizar y Mariluz Guenaga (2019). Creativity in the acquisition of computational thinking. Interactive Learning Environments, 27:5-6, 628-644, DOI: 10.1080/10494820.2019.1610451

Otros artículos y participaciones en congresos relacionados con este trabajo:

- Hershkovitz, A., Sitman, R., Israel-Fishelson, R., Eguíluz, A., Garaizar, P. y Guenaga, M. (2019). Creativity inside and outside programming learning. En *2nd Educational Data Mining in Computer Science Education Workshop at The 9th International Conference on Learning Analytics and Knowledge (LAK '19)* (March 4-8, Tempe, AZ).
- Gal, L., Hershkovitz, A., Eguíluz, A., Guenaga, M. y Garaizar, P. (2017, April). Suggesting a Log-Based Creativity Measurement for Online Programming Learning Environment. En *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale* (pp. 273-277). ACM.
- Guenaga, M., Mentxaka, I., Garaizar, P., Eguiluz, A., Villagrasa, S. y Navarro, I. (2017, July). Make World, a collaborative platform to develop computational thinking and STEAM. En *International Conference on Learning and Collaboration Technologies (pp. 50-59)*. Springer, Cham.
- Guenaga, M., Menchaca, I., Garaizar, P. y Eguíluz, A. (2017, October). Trastea. club, an initiative to develop computational thinking among young students. En *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality* (p. 10). ACM.

Adicionalmente, se han publicado tres conjuntos de datos para su uso abierto por la comunidad investigadora. Están disponibles en el *Open Science Framework*:: https://osf.io/vvs3b/.

H.2. En proceso de elaboración

Como resultado de la investigación expuesta en esta tesis, estamos trabajando actualmente en los siguientes artículos para su publicación en revistas indexadas:

- Andoni Eguíluz, Mariluz Guenaga, Pablo Garaizar y Juanjo Gibaja (2020). "Challenges and Difficulties in the Analysis of the Development of Computational Thinking in an Online Game". Presentado a *Special Issue: Assessing Computational Thinking* de Computer Science Education, ISSN 0899-3408.
- Andoni Eguíluz, Mariluz Guenaga, Pablo Garaizar y Juanjo Gibaja (2020). "Incorporation of Generalization in Computational Thinking Tools: Does it Improve the Learners' Performance?". En preparación para presentación a *ACM Transactions on Computing Education (TOCE)*.
- Rotem Israel-Fishelson, Arnon Hershkovitz, Andoni Eguíluz, Mariluz Guenaga y Pablo Garaizar (2020). "The Associations between Computational Thinking and Creativity: The Role of Personal Characteristics", aceptado por *Journal of Educational Computing Research*, en proceso de revisión.

Bibliografía

Abrahamson, D. 2016. «The shape of things to come: The computational pictograph as a bridge from combinatorial space to outcome distribution.» International Journal of Computers for Mathematical Learning 11 1: 137-146.

Ackermann, E.K. 2012. «Programming for the natives: What is it? What's in it for the kids.» Editado por C., Clayson, J.E. y Yiannoutsou, N. Kynigos. Constructionism 2012. School of Philosophy, National & Kapodistrian University of Athens, Greece.

Aho, A. V. 2012. «Computation and computational thinking.» The Computer Journal 55 7: 832-835.

Aho, A.V. 2011. «Ubiquity symposium: Computation and computational thinking.» Ubiquity 1.

Alves, N. D. C., C. G. von Wangenheim, J. C. R. Hauck, y A. F. Borgatto. 2020. «A Large-scale Evaluation of a Rubric for the Automatic Assessment of Algorithms and Programming Concepts.» Proceedings of the 51st ACM Technical Symposium on Computer Science Education. ACM. 556-562.

Armoni, M., O. Meerbaum-Salant, y M. Ben-Ari. 2015. «From scratch to "real" programming.» ACM Transactions on Computing Education (TOCE) 14 4: 25.

Baker, R.S. 2007. «Modeling and understanding students' off-task behavior in intelligent tutoring systems.» Proceedings of the SIGCHI conference on Human factors in computing systems. ACM. 1059-1068.

Balanskat, A., y K. Engelhardt. 2015. Computer programming and coding: Priorities, school curricula and initiatives across Europe. European Schoolnet.

Barbosa, L. L. D. S., y M. V. Maltempi. 2019. «Recognizing Possibilities of Computational Thinking When Teaching First-degree Equations: A Classroom Case.» Proceedings of FabLearn 2019. ACM. 57-64.

Barr, V., y C. Stephenson. 2011. «Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community?» Acm Inroads 2 1: 48-54.

Basawapatna, A. R., A. Repenning, K. H. Koh, y H. Nickerson. 2013. «The zones of proximal flow: guiding students through a space of computational thinking skills and challenges.» Proceedings of the ninth annual international ACM conference on International computing education research. ACM. 67-74.

Basawapatna, A. R., K. H. Koh, y A. Repenning. 2010. «Using scalable game design to teach computer science from middle school to graduate school.» Proceedings of the fifteenth annual conference on Innovation and technology in computer science education ACM 224-228.

Basu, S., D. Rutstein, Y. Xu, y L. Shear. 2020. «A Principled Approach to Designing a Computational Thinking Practices Assessment for Early Grades.» Proceedings of the 51st ACM Technical Symposium on Computer Science Education. 912-918.

Bau, D. 2015. «Droplet, a blocks-based editor for text code.» Journal of Computing Sciences in Colleges 30 6: 138-144.

Bau, D., D. A. Bau, M. Dawson, y C. S. Pickens. 2015. «Pencil code: block code for a text world.» Proceedings of the 14th International Conference on Interaction Design and Children. 445-448.

Bau, D., J. Gray, C. Kelleher, J. Sheldon, y F. Turbak. 2017. «Learnable programming: blocks and beyond.» Communications of the ACM 60(6): 72-80. doi:10.1145/3015455.

Begel, A. 1996. LogoBlocks: A Graphical Programming Language for Interacting with the World. Cambridge: Electrical Engineering and Computer Science Department. MIT.

Begel, A., y E. Klopfer. 2007. «Starlogo TNG: An introduction to game development.» Journal of E-Learning 53: 146.

Bell, T., I. H. Witten, M. Fellows, R. Adams, J. McKenzie, y S. Jarman. 2015. «CS Unplugged: an enrichment and extension programme for primary-aged students.» csunplugged.org. Último acceso: 15 de 1 de 2020. https://classic.csunplugged.org/wp-content/uploads/2015/03/CSUnplugged_OS_2015_v3.1.pdf.

Bers, M. U. 2017. Coding as a Playground: Programming and Computational Thinking in the Early Childhood Classroom. Routledge press.

Bers, M., L. Flannery, E. Kazakoff, y A. Sullivan. 2014. «Computational thinking and tinkering: Exploration of an early childhood robotics curriculum.» Computers & Education, vol. 72 Computers & Education, vol. 72, pp. 145-157 145-157.

Bienkowski, M., E. Snow, D. W. Rutstein, y S. Grover. 2015. Assessment design patterns for computational thinking practices in secondary computer science: A first look. SRI International.

Blikstein, P., M. Worsley, C. Piech, M. Sahami, S. Cooper, y D. Koller. 2014. «Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming.» Journal of the Learning Sciences 23 4: 561-599.

Boe, B., C. Hill, M. Len, G. Dreschler, P. Conrad, y D. Franklin. 2013. «Hairball: Lint-inspired static analysis of scratch projects.» Proceeding of the 44th ACM technical symposium on Computer science education ACM 215-220.

Bonar, J. G., y B. W. Liffick. 1987. A Visual Programming Language for Novices. No. UPITT/LRDC/ONR/LSP-5, Pittsburgh University. Learning Research and Development Center.

Bottoni, P., y M. Ceriani. 2015. «Using blocks to get more blocks: Exploring linked data through integration of queries and result sets in block programming.» 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond). IEEE. 99-101.

Brandt, J., M. Dontcheva, M. Weskamp, y S. R. Klemmer. 2010. «Example-centric programming: integrating web search into the development environment.» Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 513-522.

Brennan, K., y M. Resnick. 2012. «New frameworks for studying and assessing the development of computational thinking.» Proceedings of the 2012 annual meeting of the American Educational Research Association 1-25.

Brown, N. C. C., M. Kölling, T. Crick, S. Peyton Jones, S. Humphreys, y S. Sentance. 2013. «Bringing computer science back into schools: lessons from the UK.» Proceedings of the 44th ACM technical symposium on Computer science education ACM 269-274.

Brusilovsky, P., E. Calabrese, J. Hvorecky, A. Kouchnirenko, y P. Miller. 1997. «Mini-languages: a way to learn programming principles.» Education and information technologies 2 1: 65-83.

Buffum, P.S., E.V. Lobene, M.H. Frankosky, K.E. Boyer, E.N. Wiebe, y J.C. Lester. 2015. «A practical guide to developing and validating computer science knowledge assessments with application to middle school.» Proceedings of the 46th ACM Technical Symposium on Computer Science Education ACM 622-627.

Buitrago Flórez, F., R. Casallas, M. Hernández, A. Reyes, S. Restrepo, y G. Danies. 2017. «Changing a generation's way of thinking: Teaching computational thinking through programming.» Review of Educational Research 87(4): 834-860.

Cartelli, A., V. Dagiene, y G. Futschek. 2010. «Bebras contest and digital competence assessment: analysis of frameworks.» International Journal of Digital Literacy and Digital Competence (IJDLDC) 1 1: 24-39.

Chaffin, A., K. Doran, D. Hicks, y T. Barnes. 2009. «Experimental evaluation of teaching recursion in a video game.» Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games. ACM. 79-86.

Chong, E.K. 2019. «Teaching and Learning Music through the Lens of Computational Thinking.» International Conference on Art and Arts Education (ICAAE 2018). Atlantis Press. 1-7.

Conway, M., S. Audia, T. Burnette, D. Cosgrove, y K. Christiansen. 2000. «Alice: Lessons learned from building a 3D system for novices.» Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 486-493.

Cooper, S., W. Dann, y R. Pausch. 2000. «Alice: a 3-D tool for introductory programming concepts.» Journal of Computing Sciences in Colleges Consortium for Computing Sciences in Colleges 15 5: 107-116.

Csizmadia, A., P. Curzon, M. Dorling, S. Humphreys, T. Ng, C. Selby, y J. Woollard. 2015. Computational thinking-A guide for teachers. Guía, Computing At School. Último acceso: 2020 de 1 de 16. http://computingatschool.org.uk/computationalthinking.

CSTA, y ISTE. 2011. «Computational thinking teacher resources.» Último acceso: 17 de 1 de 2020. http://iste.org/computational-thinking/.

Czikszentmihalyi, M. 1990. Flow: The psychology of optimal experience. . New York: Harper & Row.

Dagiene, V. 2005. «Competition in information technology: an informal learning.» EuroLogo 228-234.

Daly, T. 2009. «Using introductory programming tools to teach programming concepts: A literature review.» The Journal for Computing Teachers 1-6.

Dann, W., D. Cosgrove, D. Slater, D. Culyba, y S. Cooper. 2012. «Mediated transfer: Alice 3 to java.» Proceedings of the 43rd ACM technical symposium on Computer Science Education. ACM. 141-146.

Dasgupta, S., W. Hale, A. Monroy-Hernández, y B.M. Hill. 2016. «Remixing as a Pathway to Computational Thinking.» Editado por ACM. Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing (CSCW '16) 1438-1449.

Denner, J., L. Werner, y E. Ortiz. 2012. «Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts?» Computers & Education 58 1: 240-249.

Denning, P.J. 2017. «Remaining trouble spots with computational thinking.» Communications of the ACM 60(6): 33-39. doi:10.1145/2998438.

Denning, P.J. 2009. «The profession of IT Beyond computational thinking.» Commun. ACM 52 6: 28-30.

Denny, P., A. Luxton-Reilly, E. Tempero, y J. Hendrickx. 2011. «Understanding the syntax barrier for novices.» Proceedings of the 16th annual joint conference on Innovation and technology in computer science education. 208-212.

DiSessa, A.A. 2000. Changing minds: Computers, learning, and literacy. MIT Press.

Djambong, T., V. Freiman, S. Gauvin, M. Paquet, y M. Chiasson. 2018. «Measurement of Computational Thinking in K-12 Education: The Need for Innovative Practices.» En Digital Technologies: Sustainable Innovations for Improving Teaching and Learning, 193-222. Springer.

Donzeau-Gouge, V., G. Huet, G. Kahn, y B. Lang. 1980. Programming environments based on structured editors: The MENTOR experience. INRIA-26, Institut National de Recherche D'Informatique et D'Automatique Rocquencourt (France).

Dorling, M., y M. Walker. 2014. «Computing Progression Pathways.» Computing at School. Último acceso: enero de 2020. https://www.hoddereducation.co.uk/media/Documents/ICT/Progression_Pathways_Assessment_Framework_V1-2.pdf.

Dorn, B., y M. Guzdial. 2006. «Graphic designers who program as informal computer science learners.» Proceedings of the second international workshop on Computing education research. 127-134.

Eagle, M., y T. Barnes. 2008. «Wu's castle: teaching arrays and loops in a game.» ACM SIGCSE Bulletin ACM 40 3: 245-249.

Echeverría, L., R. Cobos, M. Morales, F. Moreno, y V. Negrete. 2019. «Promoting Computational Thinking Skills in Primary School Students to Improve Learning of Geometry.» 2019 IEEE Global Engineering Education Conference (EDUCON). IEEE. 424-429.

Eguiluz, A., M. Guenaga, P. Garaizar, y C. Olivares-Rodriguez. 2020. «Exploring the Progression of Early Programmers in a Set of Computational Thinking Challenges via Clickstream Analysis.» IEEE Transactions on Emerging Topics in Computing IEEE 8 1: 256-261. doi:10.1109/TETC.2017.2768550.

Eguíluz, A., P. Garaizar, y M. Guenaga. 2018. «An evaluation of open digital gaming platforms for developing computational thinking skills.» En Simulation and Gaming, 143-168. InTechOpen. doi:10.5772/intechopen.69391.

Eow, Y. L., y R. Baki. 2010. «Computer games development and appreciative learning approach in enhancing students' creative perception.» Computers & Education 54 1: 146-161.

Fessakis, G., V. Komis, E. Mavroudi, y S. Prantsoudi. 2018. «Exploring the Scope and the Conceptualization of Computational Thinking at the K-12 Classroom Level Curriculum.» En Computational Thinking in the STEM Disciplines, 181-212. Springer.

Feurzeig, W., S. Papert, M. Bloom, R. Grant, y C. Solomon. 1970. «Programming-languages as a conceptual framework for teaching mathematics.» ACM SIGCUE Outlook ACM 4 2: 13-17.

Fields, D., L. Quirke, J. Amely, y J. Maughan. 2016. «Combining Big Data and Thick Data Analyses for Understanding Youth Learning Trajectories in a Summer Coding Camp.» Proceedings of the 47th ACM Technical Symposium on Computing Science Education 150-155.

Franklin, D., P. Conrad, B. Boe, K. Nilsen, C. Hill, M. Len, G. Dreschler, y cols. 2013. «Assessment of computer science learning in a scratch-based outreach program.» Proceeding of the 44th ACM technical symposium on Computer science education ACM 371-376.

Futschek, G. 2006. «Algorithmic thinking: the key for understanding computer science.» International conference on informatics in secondary schools-evolution and perspectives Springer 159-168.

García-Peñalvo, F.J., A.M. Rees, J. Hughes, I. Jormanainen, T. Toivonen, y J. Vermeersch. 2016a. «A survey of resources for introducing coding into schools.» Editado por F.J. García-Peñalvo (Ed.). Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM '16). New York, NY, USA: ACM. 19-26. doi:https://doi.org/10.1145/3012430.3012491.

García-Peñalvo, F.J., J. Hughes, A. Rees, I. Jormanainen, T. Toivonen, D. Reimann, M. Tuul, y M Virnes. 2016b. «Evaluation of existing resources (study/analysis).» TACCLE3 Consortium, Belgium. doi:10.5281/zenodo.163112.

Gardner, H., y K. Davis. 2013. The app generation: How today's youth navigate identity, intimacy, and imagination in a digital world. Yale University Press.

Garlick, R., y E. Cankaya. 2010. «Using Alice in CS1: A quantitative experiment.» Proceedingsfrom ITiCSE '10: The fifteenth annual conference on Innovation and technology in computer science education. Washington, DC: ACM. 165-168.

Gautam, A., W. Bortz, y D. Tatar. 2020. «Abstraction Through Multiple Representations in an Integrated Computational Thinking Environment.» Proceedings of the 51st ACM Technical Symposium on Computer Science Education. 393-399.

Gee, J. P. 2003. What video games have to teach us about learning and literacy. NewYork: MacMillan.

Geldreich, K., A. Simon, y E. Starke. 2019. «Which Perceptions Do Primary School Children Have about Programming?» Proceedings of the 14th Workshop in Primary and Secondary Computing Education. 1-7.

Gibson, B., y T. Bell. 2013. «Evaluation of games for teaching computer science.» Proceedings of the 8th Workshop in Primary and Secondary Computing Education. 51-60.

Gouws, L. A., K. Bradshaw, y P. Wentworth. 2013. «Computational thinking in educational activities: an evaluation of the educational game light-bot.» Proceedings of the 18th ACM conference on Innovation and technology in computer science education ACM 10-15.

Graven, O., y L. MacKinnon. 2008. «Prototyping a games-based environment for learning.» E-Learn: World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education. Association for the Advancement of Computing in Education (AACE). 2661-2668.

Green, T.R. 1989. «Cognitive dimensions of notations.» Editado por A. Sutcliffe y L. Macaulay. People and computers V Cambridge University Press 443-460.

Grover, S. 2017. «Assessing algorithmic and computational thinking in K-12: Lessons from a middle school classroom.» Emerging research, practice, and policy on computational thinking. Springer. 269-288.

Grover, S. 2015. «Systems of Assessments" for deeper learning of computational thinking in K-12.» Proceedings of the 2015 annual meeting of the American educational research association. 15-20.

Grover, S., M. Bienkowski, J. Niekrasz, y M. Hauswirth. 2016. «Assessing Problem-Solving Process At Scale.» Proceedings of the Third (2016) ACM Conference on Learning @ Scale (L@S '16). New York, NY, USA: ACM. 245-248. doi:https://doi.org/10.1145/.

Grover, S., S. Basu, M. Bienkowski, M. Eagle, N. Diana, y J. Stamper. 2017. «A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments.» ACM Transactions on Computing Education (TOCE) 17 3: 1-25.

Grover, S., S. Cooper, y R. Pea. 2014. «Assessing computational learning in K-12.» Proceedings of the 2014 conference on Innovation & technology in computer science education 57-62.

Grover, S., y R. Pea. 2013. «Computational thinking in K–12: A review of the state of the field.» Educational Researcher 42 1: 38-43.

Grover, S., y S. Basu. 2017. «Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic.» Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education. ACM. 267-272. doi:https://doi.org/10.1145/3017680.3017723.

Guenaga, M., I. Mentxaka, P. Garaizar, A. Eguíluz, S. Villagrasa, y I. Navarro. 2017. «Make World, a collaborative platform to develop computational thinking and STEAM.» International Conference on Learning and Collaboration Technologies. Springer. 50-59.

Gujberova, M., y I. Kalas. 2013. «Designing productive gradations of tasks in primary programming education.» Proceedings of the 8th Workshop in Primary and Secondary Computing Education (WiPSE '13). ACM. 108-117.

Guo, P.J. 2013. «Online python tutor: embeddable web-based program visualization for cs education.» Proceeding of the 44th ACM technical symposium on Computer science education. 579-584.

Guzdial, M. 2015. Learner-centered design of computing education: Research on computing for everyone. Vol. 8(6). Synthesis Lectures on Human-Centered Informatics.

Harteveld, C., G. Smith, G. Carmichael, E. Gee, y C. Stewart-Gardiner. 2014. «A design-focused analysis of games teaching computer science.» Proceedings of Games+ Learning+ Society. 1-8.

Hartness, K. 2004. «Robocode: using games to teach artificial intelligence.» Journal of Computing Sciences in Colleges 19 4: 287-291.

Henderson, P. B., T. J. Cortina, O. Hazzan, y J. M. Wing. 2007. «Computational thinking.» ACM SIGCSE Bulletin 39 1: 195-196.

Hershkovitz, A., R. Sitman, R. Israel-Fishelson, A. Eguíluz, P. Garaizar, y M. Guenaga. 2019. «Creativity Inside and Outside Programming Learning.» Proceedings of 9th International Conference on Learning Analytics & Knowledge (LAK19). 287-292.

Holbert, N. R., y U. Wilensky. 2011. «Racing games for exploring kinematics: a computational thinking approach.» Proceedings of the 7th international conference on Games + Learning + Society Conference. 109-118.

Homer, M., y J. Noble. 2014. «Combining tiled and textual views of code.» 2014 Second IEEE Working Conference on Software Visualization. IEEE. 1-10.

Hubwieser, P., y A. Mühling. 2014. «Playing PISA with bebras.» Proceedings of the 9th Workshop in Primary and Secondary Computing Education ACM 128-129.

Hutchins, N. 2018. «A DSML for a Robotics Environment to Support Synergistic Learning of CT and Geometry.» Proceedings of International Conference on Computational Thinking Education 2018. 77-82.

ISTE. 2016. ISTE Standards for Students. Último acceso: 17 de 1 de 2020. https://www.iste.org/standards/for-students.

Jiménez-Toledo, J. A., C. Collazos, y O. Revelo-Sánchez. 2019. «Considerations for the Teaching-Learning Processes of Introductory Programming Courses: A Systematic Literature Review.» TecnoLógicas 22: 82-116.

Johnson, C., y P. Bui. 2015. «Blocks in, blocks out: A language for 3D models.» 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond). IEEE. 77-82.

Kalelioğlu, F. 2015. «A new way of teaching programming skills to K-12 students: Code. org.» Computers in Human Behavior 52: 200-210.

Kalelioglu, F., Y. Gulbahar, y V. Kukul. 2016. «A framework for computational thinking based on a systematic research review.» Baltic Journal of Modern Computing 4 3: 583-596.

Kazakoff, E., y M. Bers. 2012. «Programming in a robotics context in the kindergarten classroom: The impact on sequencing skills.» Journal of Educational Multimedia and Hypermedia 21 4: 371-391.

Kazimoglu, C., Kiernan, M., L. Bacon, y L. MacKinnon. 2012. «Learning programming at the computational thinking level via digital game-play.» Procedia Computer Science 9: 522-531.

Kazimoglu, C., M. Kiernan, L. Bacon, y L. MacKinnon. 2011. «Understanding Computational Thinking before Programming: Developing Guidelines for the Design of Games to Learn Introductory Programming through Game-Play.» International Journal of Game-Based Learning (IJGBL) 1 3: 30-52.

Kelleher, C. 2006. Motivating Programming: Using storytelling to make computer programming attractive to middle school girls. Vols. (No. CMU-CS-06-171). CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

Koedinger, K. R., R. S. Baker, K. Cunningham, A. Skogsholm, B. Leber, y J. Stamper. 2010. «A data repository for the EDM community: The PSLC DataShop.» Handbook of educational data mining 43: 43-56.

Koh, K. H., A. Basawapatna, V. Bennett, y A. Repenning. 2010. «Towards the automatic recognition of computational thinking for adaptive visual language learning.» Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on IEEE 5966.

Koh, K. H., H. Nickerson, A. Basawapatna, y A. Repenning. 2014a. «Early validation of computational thinking pattern analysis.» Proceedings of the 2014 conference on Innovation & technology in computer science education ACM 213-218.

Koh, K.H., A. Basawapatna, H. Nickerson, y A. Repenning. 2014b. «Real time assessment of computational thinking.» 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE. 49-52.

Kölling, M., N. C. Brown, y A. Altadmri. 2015. «Frame-based editing: Easing the transition from blocks to text-based programming.» Proceedings of the Workshop in Primary and Secondary Computing Education. 29-38.

Kölling, M., y J. Rosenberg. 1996. «Blue—a language for teaching object-oriented programming.» Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education. 190-194.

Koschitz, D., y E. Rosenbaum. 2012. «Exploring algorithmic geometry with'Beetle Blocks:'A graphical programming language for generating 3D forms.» Proceedings of the 15 th International Conference on Geometry and Graphics. 380-389.

Kuruvada, P., D. Asamoah, N. Dalal, y S. Kak. 2010. «The Use of Rapid Digital Game Creation to Learn Computational Thinking.» arXiv preprint arXiv:1011.4093 1-9. http://arxiv.org/abs/1011.4093.

Ladd, B., y E. Harcourt. 2005. «Student competitions and bots in an introductory programming course.» Journal of Computing Sciences in Colleges 20 5: 274-284.

Laporte, L., y B. Zaman. 2016. «Informing content-driven design of computer programming games: a problems analysis and a game review.» Proceedings of the 9th Nordic Conference on Human-Computer Interaction. 1-10.

Lee, I., F. Martin, J. Denner, B. Coulter, W. Allan, J. Erickson, J. Malyn-Smith, y L. Werner. 2011. «Computational thinking for youth in practice.» Acm Inroads 2 1: 32-37.

Lerner, S., S. R. Foster, y W. G. Griswold. 2015. «Polymorphic blocks: Formalism-inspired UI for structured connectors.» Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems. ACM. 3063-3072.

Lewis, C.M. 2010. «How programming environment shapes perception, learning and goals: logo vs. scratch.» Proceedings of the 41st ACM technical symposium on Computer science education. 346-350.

Lewis, C.M. 2011. «Is pair programming more effective than other forms of collaboration for young students?» Computer Science Education 21(2): 105-134.

Li, F. W., y C. Watson. 2011. «Game-based concept visualization for learning programming.» Proceedings of the third international ACM workshop on Multimedia technologies for distance learning. ACM. 37-42.

Liñán, L. C., y Á. A. J. Pérez. 2015. «Educational Data Mining and Learning Analytics: differences, similarities, and time evolution.» International Journal of Educational Technology in Higher Education 12 3: 98-112.

Liu, C. C., Y. B. Cheng, y C. W. Huang. 2011. «The effect of simulation games on the learning of computational problem solving.» Computers & Education 57 3: 1907-1918.

Long, J. 2007. «Just For Fun: using programming games in software programming training and education.» Journal of Information Technology Education: Research 6 1: 279-290.

Lu, J. J., y G. H. Fletcher. 2009. «Thinking about computational thinking.» Proceedings of the 40th ACM technical symposium on Computer science education 260-264.

Lui, D., J. T. Walker, S. Hanna, Y. B. Kafai, D. Fields, y G. Jayathirtha. 2019. «Communicating computational concepts and practices within high school students' portfolios of making electronic textiles.» Interactive Learning Environments 1-18.

Lye, S. Y., y J. H. L. Koh. 2014. «Review on teaching and learning of computational thinking through programming: What is next for K-12?» Computers in Human Behavior 41: 51-61.

Malan, D. J., y H. H. Leitner. 2007. «Scratch for budding computer scientists.» SIGCSE Bulletin ACM 39: 223-227.

Malliarakis, C., M. Satratzemi, y S. Xinogalos. 2014. «Educational games for teaching computer programming.» Research on e-Learning and ICT in Education. New York, NY: Springer. 87-98.

—. 2013. «Towards a new massive multiplayer online role playing game for introductory programming.» Proceedings of the 6th Balkan Conference in Informatics. 156-163.

Maloney, J. H., K. Peppler, Y. Kafai, M. Resnick, y N. Rusk. 2008. «Programming by choice: urban youth learning programming with scratch.» Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education ACM 40 1: 367-371.

Maloney, J. H., y R. B. Smith. 1995. «Directness and liveness in the morphic user interface construction environment.» Proceedings of the 8th annual ACM symposium on User interface and software technology. 21-28.

Mannila, L., V. Dagiene, B. Demo, N. Grgurina, C. Mirolo, L. Rolandsson, y A. Settle. 2014. «Computational thinking in K-9 education.» Proceedings of the working

group reports of the 2014 on innovation & technology in computer science education conference ACM 1-29.

Marshall, K.S. 2011. «Was that CT? Assessing Computational Thinking Patterns through Video-Based Prompts.» Artículo entregado para la publicación.

Matsuzawa, Y., T. Ohata, M. Sugiura, y S. Sakai. 2015. «Language migration in non-cs introductory programming through mutual language translation environment.» Proceedings of the 46th ACM Technical Symposium on Computer Science Education. 185-190.

McNerney, T.S. 1999. Tangible programming bricks: An approach to making programming accessible to everyone. Tesis doctoral, Massachusetts Institute of Technology.

Meerbaum-Salant, O., M. Armoni, y M. Ben-Ari. 2011. «Habits of programming in scratch.» Proceedings of the 16th annual joint conference on Innovation and technology in computer science education ACM 168-172.

Meerbaum-Salant, O., M. Armoni, y M. Ben-Ari. 2013. «Learning computer science concepts with scratch.» Computer Science Education 23(3): 239-264.

Mendelsohn, P., T. R. G. Green, y P. Brna. 1990. «Programming languages in education: The search for an easy start.» Psychology of programming. Academic Press. 175-200.

Miller, L. D., L. K. Soh, V. Chiriacescu, E. Ingraham, D. F. Shell, y M. P. Hazley. 2014. «Integrating computational and creative thinking to improve learning and performance in CS1.» Proceedings of the 45th ACM technical symposium on Computer Science Education ACM 475-480.

Monig, J., Y. Ohshima, y J. Maloney. 2015. «Blocks at your fingertips: Blurring the line between blocks and text in GP.» 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond). IEEE. 51-53.

Moreno-León, J., y G. Robles. 2015. «Dr. Scratch: a Web Tool to Automatically Evaluate Scratch Projects.» Editado por ACM. Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15) 132-133. doi:10.1145/2818314.2818338.

Moskal, B., D. Lurie, y S. Cooper. 2004. «Evaluating the effectiveness of a new instructional approach.» SIGCSE Bulletin (Special Interest Group on Computer Science Education) ACM 36 1: 75-79.

Mühling, A., A. Ruf, y P. Hubwieser. 2015. «Design and first results of a psychometric test for measuring basic programming abilities.» Proceedings of the workshop in primary and secondary computing education ACM 2-10.

Muratet, M., P. Torguet, y J. P. Jessel. 2009. «Learning programming with an RTS based Serious Game.» Serious Games on the Move 181-192.

National Research Council. 2010. Report of a workshop on the scope and nature of computational thinking. National Academies Press.

Palts, T., y M. Pedaste. 2020. «A Model for Developing Computational Thinking Skills.» Informatics in Education 19 1: 113-128.

Panitz, M., K. Sung, y R. Rosenberg. 2010. «Game programming in CS0: a scaffolded approach.» Journal of Computing Sciences in Colleges 26(1): 126-132.

Papastergiou, M. 2009. «Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation.» Computers & education 52(1): 1-12.

Papert, S. 1980. Mindstorms: Children, computers, and powerful ideas. Basic Books, Inc.

Parsons, D., y P. Haden. 2007. «Programming osmosis: Knowledge transfer from imperative to visual programming environments.» Procedings of The Twentieth Annual NACCQ Conference. 209-215.

Pattis, R.E. 1981. Karel the robot: a gentle introduction to the art of programming. John Wiley & Sons, Inc.

Perkins, K., W. Adams, M. Dubson, N. Finkelstein, S. Reid, C. Wieman, y R. LeMaster. 2006. «PhET: Interactive simulations for teaching and learning physics.» The physics teacher 44 1: 18-23.

Perlman, R. 1976. «Using computer technology to provide a creative learning environment for preschool children.»

Piech, C., M. Sahami, D. Koller, S. Cooper, y P. Blikstein. 2012. «Modeling how students learn to program.» Proceedings of the 43rd ACM technical symposium on Computer Science Education 153-160.

Pinto-Llorente, A.M., S. Casillas Martín, M. Cabezas González, y F.J. García-Peñalvo. 2016. «Developing computational thinking via the visual programming tool: lego education WeDo.» Editado por Francisco José García-Peñalvo (Ed.). Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM '16). New York, NY, USA: ACM. 45-50. doi:https://doi.org/10.1145/3012430.3012495.

Powers, K., P. Gross, S. Cooper, M. Mcnally, K. Goldman, V. Proulx, y M. Carlisle. 2006. «Tools for teaching introductory programming: What works?» ACM SIGCSE Bulletin 38(1): 560-561. doi:10.1145/1124706.1121514.

Powers, K., S. Ecott, y L. M. Hirshfield. 2007. «Through the looking glass: teaching CS0 with Alice.» Proceedings of the 38th SIGCSE technical symposium on Computer science education. 213-217.

Prensky, M. 2003. «Digital game-based learning.» Computers in Entertainment (CIE) 1 1: 21-21.

Reed, W. M., y D. B. Palumbo. 1988. «The Effect of the BASIC Programming Language on Problem-Solving Skills and Computer Anxiety.» Computers in the Schools 4 3-4: 91-104. doi:10.1300/j025v04n03_11.

Repenning, A. 2000. «AgentSheets®: An interactive simulation environment with end-user programmable agents.» Interaction.

Repenning, A., D. Webb, y A. Ioannidou. 2010. «Scalable game design and the development of a checklist for getting computational thinking into public schools.» Proceedings of the 41st ACM technical symposium on Computer science education. 265-269.

Repenning, A., y A. Ioannidou. 2008. «Broadening participation through scalable game design.» ACM SIGCSE Bulletin ACM 40 1: 305-309.

Resnick, M., J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, y cols. 2009. «Scratch: programming for all.» Commun. ACM ACM 52 11: 60-67.

Robertson, J., y C. Howells. 2008. «Computer game design: Opportunities for successful learning.» Computers & Education 50 2: 559-578.

Robins, A., J. Rountree, y N. Rountree. 2003. «Learning and teaching programming: A review and discussion.» Computer science education 13 2: 137-172.

Román-González, M., J.C. Pérez-González, y C. Jiménez-Fernandez. 2017. «Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test.» Computers in Human Behavior 1-14.

Roque, R.V. 2007. «OpenBlocks: an extendable framework for graphical block programming systems.» Tesis Doctoral, Massachusetts Institute of Technology.

Royal Society. 2012. «Shut down or restart? The way forward for computing in UK schools.» Último acceso: 16 de 01 de 2020. https://royalsociety.org/~/media/education/computingin-schools/2012-01-12-computing-in-schools.pdf.

Rushkoff, D. 2010. Program or be programmed: Ten commands for a digital age. Or Books.

Sabitzer, B., y H. Demarle-Meusel. 2018. «A congress for children and computational thinking for everyone.» Proceedings of the 13th Workshop in Primary and Secondary Computing Education. ACM. 25.

Sandoval-Reyes, S., P. Galicia-Galicia, y I. Gutierrez-Sanchez. 2011. «Visual learning environments for computer programming.» Electronics, Robotics and

Automotive Mechanics Conference (CERMA) IEEE 439-444. doi:10.1109/CERMA.2011.76.

Saqr, M., y M. Tedre. 2019. «Should we teach computational thinking and big data principles to medical students?» International journal of health sciences 13 4: 1-2.

Schell, J. 2008. The Art of Game Design: A book of lenses. CRC press.

Seif El-Nasr, M., I. Yucel, J. Zupko, A. Tapia, y B. Smith. 2007. «Middle-to-High School Girls as Game Designers–What are the Implications?» 2nd Annual Microsoft Academic Days Conference on Game Development.

Seiter, L., y B. Foreman. 2013. «Modeling the learning progressions of computational thinking of primary grade students.» Proceedings of the ninth annual international ACM conference on International computing education research 59-66.

Selby, C., M. Dorling, y J. Woollard. 2014. «Evidence of assessing computational thinking.»

Selby, C., y J. Woollard. 2013. «Computational Thinking: The Developing Definition.» 19th Annual Conference on Innovation and Technology in Computer Science Education. Canterbury, Great Britain.

Seoane Pardo, A.M. 2018. «Computational Thinking Between Philosophy and STEM-Programming Decision Making Applied to the Behavior of" Moral Machines" in Ethical Values Classroom.» Revista iberoamericana de tecnologías del aprendizaje - IEEE RITA IEEE 13 1: 20-29.

Slany, W. 2014. «Tinkering with Pocket Code, a Scratch-like programming app for your smartphone.» Proceedings of Constructionism 2014. Viena, Austria.

Solomon, C. J., y S. Papert. 1976. «A case study of a young child doing Turtle Graphics in LOGO.» Proceedings of the June 7-10, 1976, national computer conference and exposition. 1049-1056.

Sorva, J. 2012. Visual program simulation in introductory programming education. Aalto University.

Sorva, J., y T. Sirkiä. 2011. «Context-sensitive guidance in the UUhistle program visualization system.» Proceedings of the 6th Program Visualization Workshop (PVW'11). 77-85.

Spacco, J., P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, y R. Duvall. 2015. «Analyzing student work patterns using programming exercise data.» Proceedings of the 46th ACM Technical Symposium on Computer Science Education 18-23.

Stefik, A., y S. Siebert. 2013. «An empirical investigation into programming language syntax.» ACM Transactions on Computing Education (TOCE) 13 4: 1-40.

Su, J. M., M. J. Li, W. D. Li, y X. Y. Zhuang. 2019. «Building an Authoring Tool to Create Blockly-based Programming Learning Games for Elementary Students.» 2019 8th International Congress on Advanced Applied Informatics (IIAI-AAI). IEEE. 246-249.

Sung, K., C. Hillyard, R.L. Angotti, M.W. Panitz, D.S. Goldstein, y J. Nordlinger. 2010. «Game-Themed Programming Assignment Modules: A Pathway for Gradual Integration of Gaming Context into Existing Introductory Programming Courses.» IEEE Education Society 54 3: 416-427.

Suzuki, H., y H. Kato. 1993. «AlgoBlock: a tangible programming language, a tool for collaborative learning.» Proceedings of 4th European Logo Conference. 297-303.

Tedre, M., y P.J. Denning. 2016. «The long quest for computational thinking.» Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16). New York, NY: ACM. 120-129. doi:10.1145/2999541.2999542.

Tew, A. E., y B. Dorn. 2013. «The case for validated tools in computer science education research.» Computer 46 9: 60-66.

Tew, A. E., y M. Guzdial. 2011. «The FCS1: a language independent assessment of CS1 knowledge.» Proceedings of the 42nd ACM technical symposium on Computer science education ACM 111-116.

Tilley, E., y J. Gray. 2017. «Dronely: A Visual Block Programming Language for the Control of Drones.» Proceedings of the SouthEast Conference. ACM. 208-211.

Tinker, R. F., y Q. Xie. 2008. «Applying computational science to education: the molecular workbench paradigm.» Computing in Science & Engineering 10 5: 24.

Touretzky, D.S., C. Gardner-McCune, y A. Aggarwal. 2017. «Semantic Reasoning in Young Programmers.» Editado por ACM. Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17) 585-590. doi:10.1145/3017680.3017.

Vahldick, A., A. J. Mendes, y M. J. Marcelino. 2014. «A review of games designed to improve introductory computer programming competencies.» 2014 IEEE frontiers in education conference (FIE) proceedings. IEEE. 1-7.

Vasek, M. 2012. Representing Expressive Types in Blocks Programming Languages. Editado por Honors Thesis Collection. Vol. 24. Wellesley College. https://repository.wellesley.edu/thesiscollection/24.

Vihavainen, A., J. Airaksinen, y C. Watson. 2014. «A systematic review of approaches for teaching introductory programming and their influence on success.» Proceedings of the tenth annual conference on International computing education research ACM 19-26.

Vihavainen, A., M. Luukkainen, y P. Ihantola. 2014. «Analysis of source code snapshot granularity levels.» Proceedings of the 15th Annual Conference on Information technology education. ACM. 21-26.

Vogel, J. J., Vogel, D. S., J. Cannon-Bowers, C. A. Bowers, K. Muse, y M. Wright. 2006. «Computer gaming and interactive simulations for learning: A meta-analysis.» Journal of Educational Computing Research 34 3: 229-243.

Vygotsky, L. 1996. Thought and language. MIT press.

Wagh, A., y U. Wilensky. 2012. «Evolution in blocks: Building models of evolution using blocks.» Proc. of Constructionism. Athens, Greece.

Weintrop, D., E. Beheshti, M. Horn, K. Orton, K. Jona, L. Trouille, y U. Wilensky. 2016. «Defining computational thinking for mathematics and science classrooms.» Journal of Science Education and Technology 25 1: 127-147.

Weintrop, D., y U. Wilensky. 2012. «RoboBuilder: A program-to-play constructionist video game.» Proceedings of the Constructionism 2012 Conference. Athens, Greece.

Weintrop, D., y U. Wilensky. 2015a. «To block or not to block, that is the question: students' perceptions of blocks-based programming.» Proceedings of the 14th International Conference on Interaction Design and Children 199-208.

Weintrop, D., y U. Wilensky. 2015b. «Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs.» ICER 15: 101-110.

Werner, L., C. McDowell, y J. Denner. 2013a. «A first step in learning analytics: pre-processing low-level Alice logging data of middle school students.» Journal of Educational Data Mining (JEDM) 5 2: 11-37.

Werner, L., C. McDowell, y J. Denner. 2013b. «Middle school students using Alice: what can we learn from logging data?» Proceeding of the 44th ACM technical symposium on Computer science education 507-512.

Werner, L., J. Denner, S. Campe, y D.C. Kawamoto. 2012. «The fairy performance assessment: measuring computational thinking in middle school.» Proceedings of the 43rd ACM technical symposium on Computer Science Education 215-220.

Wilensky, U., y K. Reisman. 2006. «Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories—an embodied modeling approach.» Cognition and instruction 24 2: 171-209.

Wilkerson-Jerde, M. H., y U. Wilensky. 2010. «Restructuring change, interpreting changes: The deltatick modeling and analysis toolkit.» Proceedings of constructionism 2010. Paris, Francia.

Wilson, A., T. Hainey, y T. Connolly. 2012. «Evaluation of computer games developed by primary school children to gauge understanding of programming concepts.» European Conference on Games Based Learning. Academic Conferences International Limited. 549.

Wilson, C., L. A. Sudol, C. Stephenson, y M. Stehlik. 2010. Running on empty: The failure to teach K-12 computer science in the digital age. Association for Computing Machinery. https://runningonempty.acm.org/fullreport2.pdf.

Wing, J.M. 2006. «Computational thinking.» Communications of the ACM 49 2: 33-35. doi:10.1145/1118178.1118215.

Wing, J.M. 2011. «Research notebook: Computational thinking—What and why?» The Link Magazine Carnegie Mellon University. https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why.

Wing, J.M. 2016. «Computational thinking, 10 years later.» Microsoft Research Blog, 23. Último acceso: 16 de 1 de 2020. https://www.microsoft.com/en-us/research/blog/computational-thinking-10-years-later/.

Wing, J.M., y D. Stanzione. 2016. «Progress in computational thinking, and expanding the HPC community.» Communications of the ACM 59(7): 10-11. doi:10.1145/2933410.

Wolber, D., H. Abelson, E. Spertus, y L. Looney. 2011. App Inventor: Create your own Android Apps. O'Reilly Media.

Wolz, U., C. Hallberg, y B. Taylor. 2011. «Scrape: A tool for visualizing the code of Scratch programs.» Poster presented at the 42nd ACM Technical Symposium on Computer Science Education. Dallas, TX.

Wolz, U., H. H. Leitner, D. J. Malan, y J. Maloney. 2009. «Starting with scratch in CS 1.» Proceedings of the 40th ACM technical symposium on Computer science education. 2-3.

Wu, W. H., H. C. Hsiao, P. L. Wu, C. H. Lin, y S. H. Huang. 2012. «Investigating the learning-theory foundations of game-based learning: a meta-analysis.» Journal of Computer Assisted Learning 28 3: 265-279.

Yeh, K.C. 2009. «Using an educational computer game as a motivational tool for supplemental instruction delivery for novice programmers in learning computer programming.» Society for Information Technology & Teacher Education International Conference. Association for the Advancement of Computing in Education (AACE). 1611-1616.

Yu, J., y R. Roque. 2018. «A survey of computational kits for young children.» Proceedings of the 17th ACM Conference on Interaction Design and Children. ACM. 289-299.

Zelkowitz, M.V. 1973. «Reversible execution.» Communications of the ACM ACM 16 9: 566.

Zur-Bargury, I. 2012. «A new curriculum for junior-high in computer science.» Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education 204-208.

Zur-Bargury, I., B. Pârv, y D. Lanzberg. 2013. «A nationwide exam as a tool for improving a new curriculum.» Proceedings of the 18th ACM conference on Innovation and technology in computer science education 267-272.