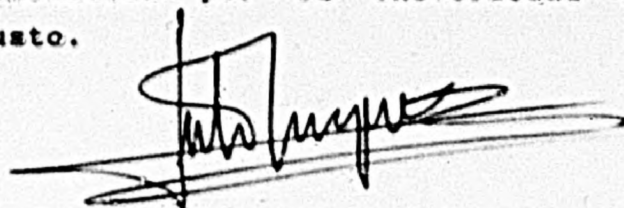
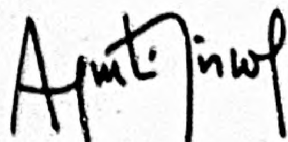
ALFA-BETA DEDUCCIÓN

Memoria presentada por
Julio Márquez González de Audicana
para optar al grado de Doctor en
Informática por la Universidad de
Deusto.

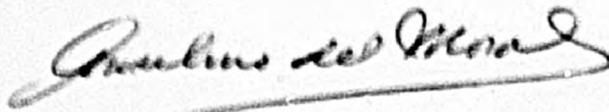


Bilbao, febrero de 1.991.

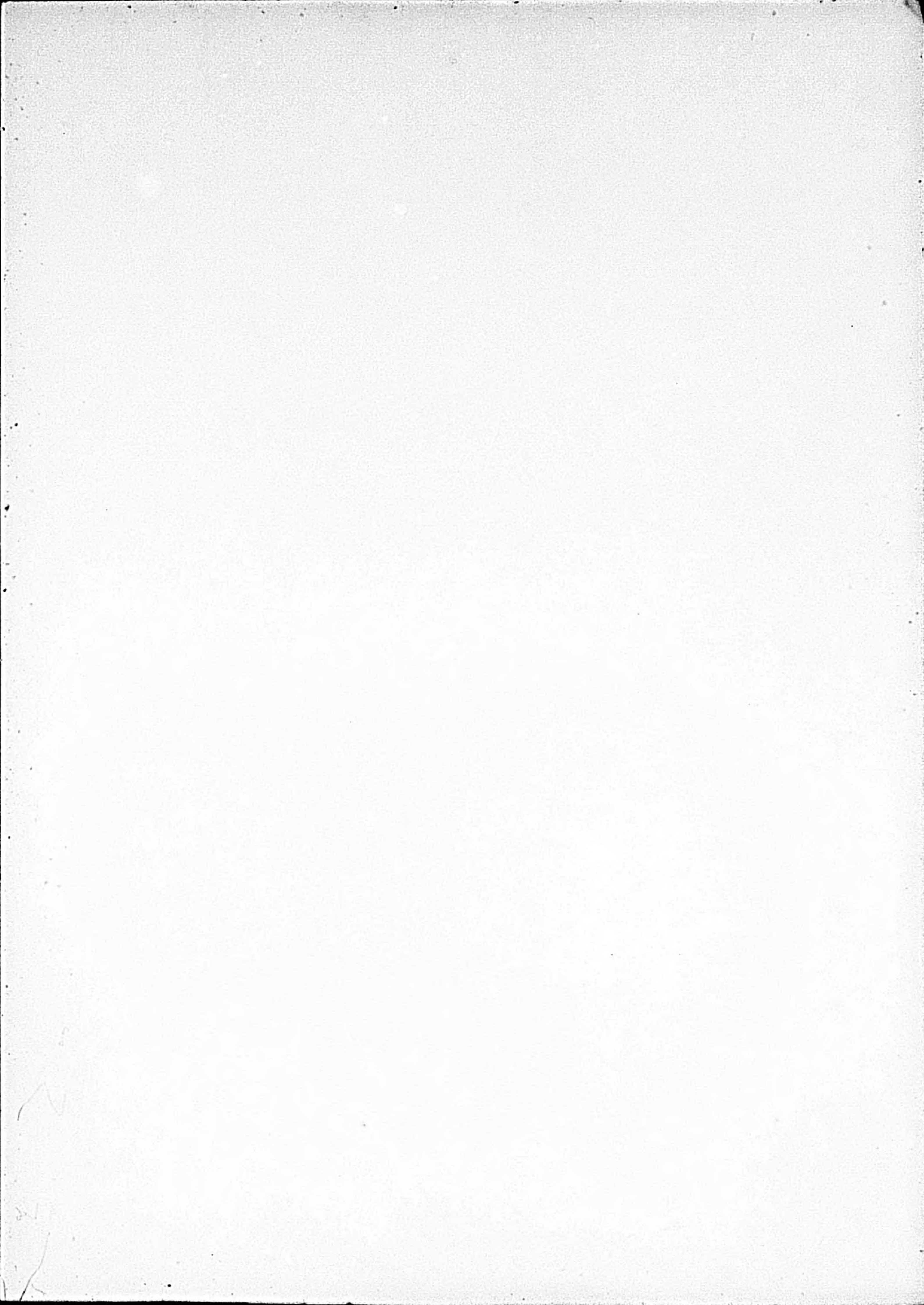
Director de la Memoria:
Prof. Dr. D. Agustín Riscos Fernández



V°B° Prof. Tutor y Director del Dpto.:
Prof. Dr. D. Anselmo del Moral Bueno



DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN



Deseo expresar mi agradecimiento a todos los que me han ayudado durante la realización de este trabajo tanto en Sevilla como en Bilbao; deseo citar especialmente al Profesor Agustín Riscos Fernández por su dirección. También quiero dedicar este trabajo a Cristina Haya Clauss.

INDICE:

INTRODUCCION

CAPITULO 0.- ESTUDIO DE UN JUEGO.

1.- Descripción del juego.
2.- Estrategia para el juego.
3.- Conclusión.

CAPITULO I.- FORMALIZACION BASICA.

1.- Preliminares.
2.- Juegos.

CAPITULO II.- GRAFOS Y/O.

1.- Introducción.
2.- Definición de grafo Y/O. Teorema de equivalencia con juegos.
3.- Arboles Y/O. Arboles Alternados.

CAPITULO III.- PROCEDIMIENTOS MIN-MAX Y ALFA-BETA.

1.- Procedimiento Min-Max.
2.- Procedimiento Alfa-Beta.

CAPITULO IV.- α-β DEDUCCION.

1.- Planteamiento.
2.- Juego deductivo.
3.- α-β Deducción.
4.- Ejemplos.

CONCLUSIONES

BIBLIOGRAFIA

INTRODUCCION

La presente memoria se enmarca dentro del campo del Razonamiento Automático, una de las áreas más puras de la Inteligencia Artificial. Su objetivo final consiste en la aplicación de estrategias de juegos a la demostración automática de fórmulas a partir de otras, esto es, a resolver problemas de deducción.

En Banerji [BA], pág. 69, puede leerse:

> *In connection with what is called the "problem-reduction" approach to problem solving ... Nilsson's analysis was made not on games but on what Amarel has called "and-or" trees. Sagle has informally argued that these are equivalent to games...This conjecture, unfortunately, is informal...*

(ver [BA], [NI], y sus referencias bibliográficas).

Podemos decir que este párrafo es el origen de la memoria. Nos planteamos las preguntas:

¿Cuál es la relación entre juegos y árboles "and/or"? (escribiremos Y/O por "and/or")

Dado que los problemas deductivos expresados con cláusulas de Horn tienen asociado un grafo Y/O (lo que ha dado lugar al desarrollo de estrategias de búsqueda en grafos), ¿cuáles son las relaciones entre grafos y árboles Y/O?, y, lo que es más interesante, ¿cómo están relacionados los juegos con los problemas deductivos?

La primera dificultad consistió en la generalización de lo expresado por Banerji: en la mayor parte de la literatura todos estos conceptos aparecen de manera informal. Por ejemplo, se supone que el concepto de *grafo* es preciso, a partir de ejemplos gráficos, que el concepto de *juego* es trivial, que el paso de grafo a árbol (necesario para los procedimientos min-max y α-β) es elemental (basta "repetir las veces que sea necesario los nodos convenientes"), y, por último, las descripciones de min-max y α-β son cualquier cosa menos descripciones de algoritmos en pseudo-código.

Esta ausencia de formalismo adecuado hace extraordinariamente difícil la búsqueda de relaciones y, sobre todo, la demostración de cualquier tipo de resultados.

Nuestra formación matemática implicaba inevitablemente la convicción de que, para poder responder a las preguntas planteadas era precisa la formalización adecuada de todos los aspectos del problema, así como la demostración de todos los resultados afirmados.

Por otra parte, nuestra formación informática nos llevaba por un lado a no aceptar resultados o procedimientos no constructibles, es decir, que no puedan de hecho ser programados en un ordenador, y por otro a aceptar como válidos los conceptos definidos constructivamente, mediante la descripción precisa del algoritmo de construcción.

Citaremos aquí que hemos programado todos los algoritmos que se usan, y los programas se incluyen en la memoria. Los programas se han realizado en LISP (Le_Lisp, versión para PC).

Estos programas se utilizan para comprobar la corrección de algoritmos, aclarar conceptos y para disponer de ejemplos; como decimos más adelante, no se ha estudiado la complejidad, lo que puede hacer que, desde un punto de vista operativo, algunos sean innecesarios. Esto no nos ha preocupado pues, en este caso, nos interesaba mucho más la claridad en la ilustración de conceptos que la eficacia; y por otra parte, ¡los programas funcionan correctamente!

La memoria tiene dos partes claramente diferenciadas:

Estudio de las relaciones grafos-juegos

Se definen formalmente juegos y grafos, se estudian sus propiedades, y se llega al resultado crucial de esta parte (teorema II.2.10) que nos afirma que en un juego existe una estrategia ganadora para un estado del juego, si y solamente si, en un grafo, asociado al juego, existe un g-camino desde el estado dado al conjunto de estados ganadores.

Aplicación de estrategias de juegos a la deducción

Resolvemos aquí el problema inverso (en cierto sentido) del anterior: dado un problema deductivo (y por tanto su grafo Y/O asociado), construimos un juego de forma que el teorema anterior se satisfaga. LLegamos así al resultado de que el problema de deducción propuesto se resuelve afirmativamente, si y solamente si, existe estrategia ganadora para el juego. Por último, se aplica la técnica α-β para la determinación de esta estrategia.

Comentaremos algo más en detalle estos aspectos, describiendo por separado el contenido de cada capítulo.

En el Capítulo 0 describimos un juego, tomado de Gaskell y Whinihan. Los juegos han sido muy estudiados por los investigadores en Inteligencia Artificial, creemos que fundamentalmente porque son muy fácilmente formalizables, y porque presentan grados de complejidad tan grandes como se desee. De hecho, es uno de los tópicos más comunes en los libros sobre Inteligencia Artificial.

A lo largo de esta memoria se hace un uso muy frecuente de los conceptos de *juego* y de *estrategia ganadora* para un juego. Estos conceptos, en apariencia frívolos, pueden llegar a originar serios problemas matemáticos, de modo que la reducción de un problema de deducción a la existencia de una estrategia ganadora no será una simplificación del problema, sino un nuevo enfoque, en el que será posible trabajar con técnicas diferentes.

El juego que se presenta es un claro ejemplo de lo expuesto: se trata de un juego de aspecto absolutamente inocente, de forma que el descubrimiento de una estrategia ganadora aparenta ser un trabajo sencillo. Sin embargo, como ponemos de manifiesto, la estrategia no tiene nada de trivial, y la demostración del resultado es bastante complicada (y sorprendente).

El Capítulo I, titulado "Formalización básica", presenta las primeras definiciones y resultados en lo que se refiere a grafos y juegos. Si bien el punto de partida fue el adoptado por

[BA], en seguida vimos que debíamos encontrar nuestra propia formalización pues la de Banerji resultaba inadecuada (sus objetivos eran distintos).

Así, definimos grafo, núcleo, juego, estrategia (ganadora, evaluadora y cauta), conjunto evaluador del juego, juego interpretable en grafos, grafo de un juego, isomorfismos de juegos, etc. Destacamos aquí:

I.1.2.- *Definición.*

Un *Grafo* es un par $\langle N,S\rangle$ donde N es un conjunto (no vacío) y $S \subseteq N{\times}N$, tales que:

G.1.- La relación en N, que notamos por Ω, y que está definida por:

$n\Omega m \iff \exists k \geq 1 \;,\; \exists n_0,\ldots,n_k \in N$

$n_0 = n \;\wedge\; n_k = m \;\wedge\; [1 \leq i \leq k \Rightarrow n_i \in S(n_{i-1})]$

es un orden parcial.

G.2.- $\forall n \in N \;\; [S(n) \text{ finito}]$.

G.3.- $\exists n \in N \;\; [S^{-1}(n) = \emptyset]$, llamado nodo inicial ó raíz.

esto es, consideramos grafos sin ciclos, sucesor-finitos, y con un solo nodo inicial.

I.1.6.- *Definición.*

Un subconjunto $K \subseteq N$ se llama un *Núcleo* del grafo $\langle N,S\rangle$, si cumple: $\forall n \in N \;\; [n \in K \iff S(n) \cap K = \emptyset]$

I.2.1.- *Definición.*

Un *Juego* es una 5-tupla $\langle E,R,T,G,P\rangle$, donde

E es un conjunto finito no vacío

T, G y P son subconjuntos de E

$R \subseteq E{\times}E$ tales que:

J.1.- $(G \cup P) \cap R^{-1}[E] = \emptyset$

J.2.- $P \subseteq T$

J.3.- $G \cap T = \emptyset$

J.4.- $\forall s,t \in E \;\; [sRt \Rightarrow (s \in T \iff t \notin T)]$

J.5.- $\langle E,R\rangle$ es un grafo.

E representará el conjunto de estados del juego; G y P son los estados ganadores y perdedores respectivamente, T el conjunto de estados en los que nos toca "mover", y R es la relación binaria que determinan las reglas del juego. Como es natural, la justificación de todas estas condiciones de las definiciones se halla en el texto.

I.2.4.- *Definición.*

Dado un juego, una *Estrategia* Q es una función parcial, $Q: T \longrightarrow E$, tal que:

E.1.- $Q(s) = t \Rightarrow sRt$

E.2.- $[Q(s) = t \wedge tRu \wedge R(u) \neq \emptyset] \Rightarrow \exists v \in E \ [Q(u) = v]$

La primera condición garantiza que Q realiza sólo movimientos permitidos, y la segunda nos garantiza que Q está definida para cualquier respuesta del contrincante.

Además de formalizar con este estilo todos los conceptos, analizamos y demostramos muchas propiedades. Algunas de estas propiedades ya eran conocidas (aunque aquí adopten una nueva presentación), pero otras creemos que son originales. Destacamos:

I.2.33.- *Teorema.*

Sea $\langle E,R,T,G,P\rangle$ un juego, y sea K el núcleo de $\langle E,R\rangle$. Se tiene:

$\forall s \ [s \in K \cap T \Rightarrow$ no existe estrategia ganadora para s].

I.2.34.- *Teorema.*

$\forall s \ [s \in T - K \Rightarrow$ existe estrategia *no-perdedora* para s]

esto es, es posible evitar que gane el contrincante.

I.2.35.- *Corolario.*

$E - R^{-1}[E] = G \cup P \Rightarrow$

$\forall s \in T - K$ existe estrategia ganadora para s.

I.2.36.- *Teorema.*

$$E - R^{-1}[E] = G \cup P \Rightarrow T - K = V.$$

resultados que caracterizan, en ciertos casos, la existencia o no existencia de estrategia ganadora. El teorema I.2.36 presenta una relación, que no conociamos, entre el núcleo y el conjunto evaluador, para el caso en que el juego no admita empates.

El Capítulo II presenta ya conceptos y resultados especificamente diseñados para nuestros fines. Comenzamos con

II.2.1.- *Definición.*

Un *Grafo* Y/O es un par $\langle N,S\rangle$ donde N es un conjunto finito no vacío y $S \subseteq N \times \mathcal{P}(N)$ tales que:

i) La relación que notamos σ, definida en N por

$$n \sigma m \iff \exists k \geq 1 \;\; \exists\, n_0, n_1, \ldots, n_k \in N$$

$$n_0 = n \;\wedge\; n_k = m \;\wedge\; [1 \leq i \leq k \Rightarrow n_i \in \bigcup S(n_{i-1})]$$

es un orden parcial.

ii) $\exists! n \in N$ llamado nodo raiz ó inicial, tal que

$$\forall m \in N \quad , \quad n \notin \bigcup S(m).$$

Hemos impuesto la finitud de N pues, para los casos de grafos asociados a un problema de deducción, N será siempre finito. La condición (ii) no necesita explicación, y la (i) nos sirve para imponer que en el grafo no haya ciclos: desde el punto de vista lógico, esto implica la no existencia de autorreferencias, esto es, para demostrar una fórmula nunca tendremos que demostrar la misma fórmula.

En los grafos Y/O, el concepto de camino pierde su sentido; ahora se sustituye por:

II.2.8.- *Definición.*

Sea $\langle N,S\rangle$ un grafo Y/O. Un *g-camino* de $n \in N$ a $D \subseteq N$ es un grafo Y/O $\langle N',S'\rangle$, definido de forma recursiva por:

i) Si $n \in D$, entonces $N' = \{n\}$, $S' = \emptyset$

ii) Si $n \notin D$:

ii.1) Si existe un k-arco $\langle n, \{n_1,\ldots,n_k\}\rangle$ tal que

para cada i, $1 \le i \le k$, exista un g-camino $\langle N'_i, S'_i \rangle$ de n_i a D, entonces

$$N' = \{n\} \cup (\bigcup_{i=1}^{k} N'_i)$$

$$S' = \{\langle n, \{n_1, \ldots, n_k\} \rangle\} \cup (\bigcup_{i=1}^{k} S'_i)$$

ii.2) Si no, no existe g-camino desde el nodo n al conjunto de nodos D.

Desde el punto de vista lógico, un g-camino de n a D es una demostración de la fórmula n tomando las fórmulas de D como hipótesis, y usando los conectores como reglas de inferencia.

El teorema

II.2.10.- *Teorema.*

Sea $\langle E,R,T,G,P \rangle$ un juego. Consideramos el grafo Y/O $\langle E,S \rangle$ dado por:

$$s \in T \;:\; S(s) = \{\{n\} : n \in R(s)\}$$
$$s \notin T \;:\; S(s) = \{R(s)\}.$$

$\forall s \in T$ se tiene que: existe una estrategia ganadora para s sii existe un g-camino en $\langle E,S \rangle$, de s a G.

nos da un primer resultado orientado a nuestros objetivos. En él se especifica una fuerte relación entre un concepto relativo a los juegos y otro relativo a grafos Y/O.

La relación con los árboles se estudia a continuación.

II.3.1.- *Definición.*

Sea $\langle N,S \rangle$ un grafo. Se dice que $\langle N,S \rangle$ es un árbol si

$$\forall m \in N \; [m \neq \text{nodo raíz} \Rightarrow \dot{\exists} n \in N \; [nSm]]$$

Análogamente, si $\langle N,S \rangle$ es un grafo Y/O, se dice que $\langle N,S \rangle$ es un árbol Y/O si

$$\forall m \in N \; [m \neq \text{nodo raíz} \Rightarrow \dot{\exists} n \in N \; [m \in \bigcup S(n)]]$$

Tal nodo n se llama antecesor inmediato o *padre* del nodo m.

II.3.2.- *Definición.*

Sea $\langle N,S\rangle$ un árbol ó un árbol Y/O. La profundidad es una función $Prof : N \longrightarrow \mathbb{N}$, con $\mathbb{N}$ el conjunto de los números naturales, definida recursivamente por

$$Prof(n) = \begin{cases} 0 \quad \text{si} \quad n = \text{nodo raíz de } \langle N,S\rangle \\ 1 + Prof(\text{padre de } n) \quad \text{en otro caso} \end{cases}$$

II.3.3.- *Definición.*

Decimos que el par $\langle N,S\rangle$ es un *Arbol de juego ó Arbol Alternado*, si $\langle N,S\rangle$ es un árbol Y/O y además:

$\forall n \in N$ [Prof(n) es par $\Rightarrow$ $\forall\langle n,A\rangle \in S$ [A es unitario]], y

$\forall n \in N$ [Prof(n) es impar $\Rightarrow$ $\exists! A \in \mathcal{P}(N)$ [$\langle n,A\rangle \in S$]].

Informalmente, esto quiere decir que
$\forall n \in N$ si Prof(n) es par, entonces n está unido con sus inmediatos sucesores exclusivamente por 1-arcos, y
si Prof(n) es impar, entonces n está unido a sus inmediatos sucesores exclusivamente por un k-arco.

Este concepto, que hemos denominado árbol alternado, ha resultado fundamental en nuestro trabajo. Resulta ser una formalización muy adecuada para expresar la alternancia implícita que se da en el desarrollo de un juego: en cada nivel de profundidad la estrategia cambia, pues evidentemente, lo que es más favorable para uno de los jugadores resulta lo más desfavorable para el otro. Por otra parte, los árboles alternados nos permiten formalizar con una gran claridad los procedimientos min-max y α-β del Capítulo III.

A continuación, dado un grafo Y/O $\langle E,S\rangle$, y un subconjunto de nodos D, construimos, mediante los algoritmos GRA-ARB y AR-AA, lo que llamamos árbol Y/O y árbol Y/O alternado asociados al grafo dado, con objeto de acercarnos a la situación de los juegos. Naturalmente, hay que seguir conservando la propiedad enunciada en II.2.10, y, en efecto, si $\langle E',S'\rangle$, D' y $\langle E'',S''\rangle$, D'' son, respectivamente, el árbol Y/O y el árbol Y/O alternado (con las

transformaciones que los algoritmos determinan sobre el conjunto D), asociados al grafo Y/O <E,S>, de nodo inicial NI, se demuestra que

II.3.10.- *Teorema.*

Las afirmaciones siguientes son equivalentes:

(1) $\exists$ g-camino de NI a D en <E,S>

(2) $\exists$ g-camino de <NI,1> a D' en <E',S'>

(3) $\exists$ g-camino de <NI,1> a D'' en <E'',S''>.

El Capítulo III presenta la formalización de los procedimientos min-max y α-β. De acuerdo con nuestros objetivos, dado un juego <E,R,T,G,P>, se considera el grafo Y/O <E,S>, tal como lo describe el teorema II.2.10, y los procedimientos se describen sobre el árbol alternado <E'',S''> asociado a <E,S>. Con más precisión, lo que se utiliza es un árbol alternado, subárbol del anterior, que hemos denotado por J(s,cota), pues toma un nodo $s \in T$ como nodo inicial, y se desarrolla sólo hasta una cierta cota de profundidad.

De esta forma, la descripción de la evaluación estática de los nodos terminales (mediante una función h), y la propagación de esta evaluación, se hace de una manera muy simple:

$$EV(n) = \begin{cases} h(n) \text{ si } S''(n) = \emptyset \text{ ó si la profundidad de n en } J(s,cota) \text{ es igual a cota} \\ \max \{\min \{EV(m) : m \in A\} : \langle n,A \rangle \in S''\} \text{ en otro caso} \end{cases}$$

La descripción completa de los algoritmos es difícil de resumir aquí en unas cuantas líneas; diremos simplemente que hemos incluido tanto una versión específicamente adaptada a nuestros propósitos, como una versión general (del procedimiento min-max y del α-β) apta para su uso independiente. De hecho, se incluye la codificación de los procedimientos para un juego de pequeña complejidad, pero no trivial, que hemos llamado Tres en Raya Generalizado, con Deslizamiento, descrito en III.1.4.

En el Capítulo IV es donde convergen todos los resultados del trabajo. En el §.1. se muestra cómo, a partir de un problema de deducción (CL,D) ⊨ A, donde CL es un conjunto de cláusulas, D son los datos, y A es la pretendida conclusión, se pasa a un árbol Y/O alternado, J(A) = <E,S>, que es una pequeña modificación del árbol Y/O alternado asociado al grafo Y/O del problema. Esta pequeña modificación no es más que un ajuste técnico de la profundidad de los nodos sin sucesores, pero que se aprovecha para construir dos conjuntos G y P, que contendrán, respectivamente, los nodos provenientes del tratamiento de los datos, y de los demás nodos sin sucesores. La idea es bien simple: Los nodos de G (estados "ganadores") lo son porque, si la demostración termina en ellos, entonces hemos "ganado" el juego, es decir, hemos terminado la demostración con éxito. Sin embargo, si el proceso tropieza con nodos que no son datos, tales que ninguna regla puede aplicárseles, entonces la demostración no puede realizarse, esto es, hemos "perdido" el juego.

El paso del problema de deducción (CL,D) ⊨ A al árbol J(A) = <E,S> se justifica demostrando que

(CL,D) ⊨ A ⟺ existe g-camino de <A,1> a G en J(A).

En el §.2. se define

IV.2.1.- *Definición*

Dado un problema de deducción como el descrito, esto es, dados J(A) = <E,S>, G y P, se llama *juego deductivo asociado* al problema, el juego J_A = <E,R,T,G,P>, donde T es el conjunto de los nodos de profundidad par, y R viene dada por:

$$\forall s \in E \quad [R(s) = \bigcup S(s)].$$

En estas condiciones, demostramos el resultado fundamental de este trabajo:

IV.2.3.- *Teorema*

(CL,D) ⊨ A ⟺ existe estrategia ganadora para <A,1> en J_A.

Por consiguiente, hemos encontrado un problema equivalente al problema de deducción: la búsqueda de una estrategia ganadora

para un juego. A la vista de esto cabe preguntarse sobre el sentido que puede tener la aplicación de la técnica α-β al juego J_A. Precisamente de esta aplicación se ocupa el §.3.

En primer lugar, α-β no proporciona una estrategia ganadora garantizada, sino que nos da un movimiento que es el mejor, con relación a la función de evaluación empleada, y a la cota de desarrollo utilizada. Por tanto, a la hora de la aplicación a la demostración, es preciso efectuar una reiteración de los procesos.

Además, la ausencia de garantía citada tiene una interpretación lógica muy clara: se produce un proceso de semidecisión, pues, si esta reiteración produce la demostración (esto es, si la estrategia conduce a la victoria en el juego), es claro que el problema de deducción se resuelve afirmativamente, pero si no es así (esto es, si perdemos la partida), entonces no podemos decir con seguridad que el problema de deducción se resuelve negativamente. La enorme ventaja estriba en la esencia de la técnica α-β: en los casos en que la demostración se realice, ésta se habrá hecho con un coste mucho menor en tiempo y memoria que con cualquier método exhaustivo.

Aunque en el texto hacemos un análisis sobre la consideración de las funciones de evaluación, para un problema abstracto la solución es trivial: los nodos terminales (que son sobre los que hay que definir la función de evaluación) son, bien de G, bien de P; en ambos casos su consideración lógica es clara: los de G son ciertos por hipótesis, y los de P son indemostrables. Así, para definir la función de evaluación bastan dos valores, uno común para los nodos de G, y otro para los de P.

Finalmente, el §.4. presenta varios ejemplos de aplicación de esta técnica, que hemos llamado *α-β deducción*.

CAPITULO 0.- ESTUDIO DE UN JUEGO.

Ø.1.- DESCRIPCION DEL JUEGO.

Ø.2.- ESTRATEGIA PARA EL JUEGO.

Ø.3.- CONCLUSION.

§.1.- DESCRIPCION DEL JUEGO.

Describiremos un juego para dos jugadores, debido a Gaskell, R.E. y Whinihan, M.J. (Fibonacci Quarterly 1, 1963). Las reglas del juego son bien simples: de una pila inicial de objetos, el primer jugador retira la cantidad que desee, pero no el total. A partir de este momento, los jugadores se van alternando, retirando cada uno de ellos la cantidad que desee, pero no más del doble de lo que retiró el anterior. Gana el juego quien retire los últimos objetos. Naturalmente, se pretende diseñar una estrategia ganadora para el jugador que comienza el juego.

Un primer resultado resulta evidente: si un jugador retira una cantidad superior al tercio de los objetos de la pila, entonces el otro jugador gana en la siguiente jugada, pues puede retirar los objetos restantes.

Veamos el desarrollo de una partida muy simple, con 15 objetos en la pila inicial.
El jugador A podrá tomar (si quiere evitar que B gane a la siguiente jugada) 1,2,3, ó 4 objetos. Si por ejemplo, A toma 4 quedan 11.
El jugador B no podrá retirar más de 8 objetos (el doble de lo último retirado por A), pero por la misma razón que antes, solo podrá tomar 1,2, ó 3 objetos. Si por ejemplo, B toma 3 quedan 8.
Sin más explicaciones:

A toma 2 y quedan 6;
B toma 1 y quedan 5;
A toma 1 y quedan 4;
B toma 1 y quedan 3;
A toma 1 y quedan 2;
B toma 2 y gana.

Como se verá, a pesar de la simplicidad del juego, el encontrar una estrategia ganadora se revela extraordinariamente difícil. Si uno intenta dibujar el grafo correspondiente a la situación anterior con 15 objetos (ver figura 2.1), se dará cuenta de la enorme dificultad de hacer algo parecido con 1000 objetos.

Expondremos en los siguientes puntos del capítulo la elegante y sorprendente estrategia de los autores citados al principio.

§.2.- ESTRATEGIA PARA EL JUEGO.

Es conocida la llamada sucesión de Fibonacci:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+2} = F_{n+1} + F_n \quad \text{para} \quad n \geq 0$$

que nos da los números:

$F_0 = 0$ $\quad F_1 = 1$ $\quad F_2 = 1$ $\quad F_3 = 2$ $\quad F_4 = 3$

$F_5 = 5$ $\quad F_6 = 8$ $\quad F_7 = 13$ $\quad F_8 = 21$ $\quad F_9 = 34$ ...etc.

Es fácil convencerse, por un razonamiento inductivo, de que todo número natural $n > 0$ puede escribirse, de manera única, como suma de números de Fibonacci no consecutivos:

$$n = F_{k_1} + \ldots + F_{k_r}, \quad \text{con} \quad k_i > k_{i+1} + 1 \quad \text{y} \quad k_r > 1.$$

En efecto, no hay más que tomar como F_{k_1} el mayor número de Fibonacci que sea menor o igual que n; el siguiente sumando no será el número consecutivo de Fibonacci puesto que se tiene

$$n - F_{k_1} < F_{k_1 - 1}$$

ya que en caso contrario:

$$n - F_{k_1} \geq F_{k_1 - 1} \quad \Rightarrow \quad n \geq F_{k_1} + F_{k_1 - 1} = F_{k_1 + 1}$$

lo que contradice la hipótesis de ser F_{k_1} el mayor número de Fibonacci menor o igual que n.

En consecuencia, repitiendo el proceso con $n - F_{k_1}$ se obtiene el resultado deseado.

La unicidad viene dada por la forma en que se realiza la descomposición.

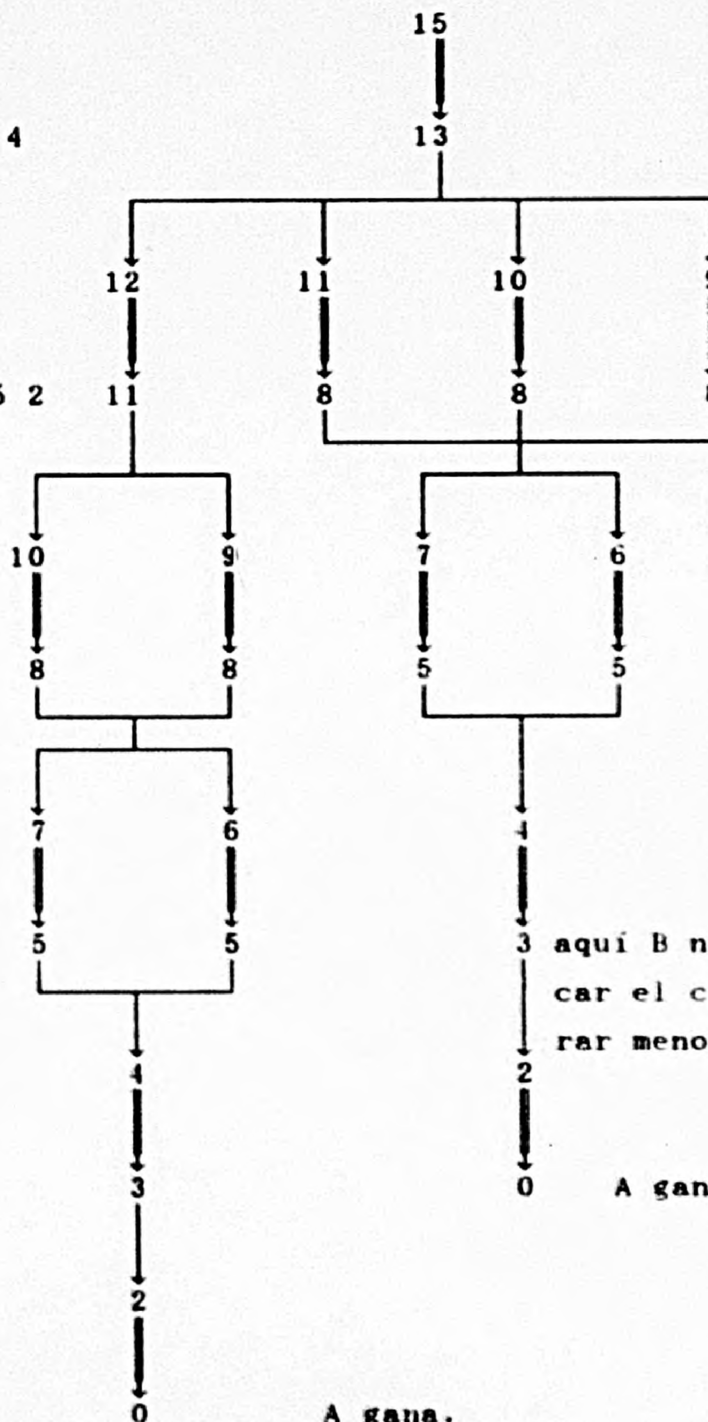
2.1.- *Definición.*

Dado un número natural $n > 0$, llamamos $\mu(n)$ al menor número de Fibonacci que aparece en su descomposición tal como se ha descrito. En la notación anterior, $\mu(n) = F_{k_r}$.

Pues bien, demostraremos a continuación que una estrategia ganadora para el juego consiste en tomar $\mu(n)$ objetos si la pila tiene n objetos. Esta estrategia se seguirá siempre, a menos que sea posible tomar los n objetos que tenga la pila y en consecuencia ganar la partida.

La partida quedaría:

Figura 2.1.



En el ejemplo anterior (ver figura 2.1):

$$15 = F_7 + F_3 = 13 + 2, \text{ teniéndose } \mu(15) = 2$$

luego, según nuestra estrategia, el jugador A toma 2 objetos y deja 13. Nótese que como $\mu(13) = 13$, el jugador B no puede seguir la misma estrategia. Esto es general por lo siguiente: si $n = F_{k_1} + \ldots + F_{k_r}$ tomamos $\mu(n) = F_{k_r}$, con lo que nos quedamos con $n' = F_{k_1} + \ldots + F_{k_{r-1}}$ y no se puede tomar $\mu(n') = F_{k_{r-1}}$ pues $F_{k_{r-1}} > 2 \cdot F_{k_r}$ al no ser consecutivos (como veremos en 2.2), con lo que tal jugada iría contra las reglas del juego.

Demostraremos que esta estrategia es ganadora con varios resultados previos.

2.2.-*Lema.*

$$n \neq \mu(n) \Rightarrow \mu(n - \mu(n)) > 2 \cdot \mu(n)$$

Prueba:

$n \neq \mu(n)$ nos dice que n no es un número de Fibonacci.

Si $n = F_{k_1} + \ldots + F_{k_r}$ será $\mu(n) = F_{k_r}$

Nótese que

$$k_{r-1} > (k_r + 1) \Rightarrow \begin{cases} k_{r-1} \geq 3 \\ (k_{r-1} - 1) > k_r \\ (k_{r-1} - 2) \geq k_r \end{cases}$$

de donde:

$$\mu(n - \mu(n)) = F_{k_{r-1}} = F_{k_{r-1}-1} + F_{k_{r-1}-2} > 2 \cdot F_{k_r} = 2 \cdot \mu(n). \blacksquare$$

2.3.- *Lema.*

$$0 < m < \mu(n) \Rightarrow \mu(m) \leq 2 \cdot (\mu(n) - m)$$

Prueba:

Sean $\mu(m) = F_j$, $\mu(n) = F_k$; como

$\mu(m) \leq m < \mu(n)$, será $F_j < F_k$

m será una suma de elementos no consecutivos que serán números de Fibonacci de entre los números F_{k-1}, F_{k-2}, ..., F_{j+1}, F_j ;

luego

$$m \le F_{k-1} + F_{k-3} + \ldots + \begin{cases} F_j \\ \text{ó} \\ F_{j+1} \end{cases} \Rightarrow \quad m + \begin{cases} F_{j-1} \\ \text{ó} \\ F_j \end{cases} \le F_k \quad ; \text{como}$$

$$F_j = F_{j-1} + F_{j-2} \le 2 \cdot F_{j-1} \qquad \text{será:}$$

$$m + \frac{1}{2} F_j \le m + \begin{cases} F_{j-1} \\ \text{ó} \\ F_j \end{cases} \le F_k \Rightarrow \quad F_j \le 2 \cdot (F_k - m) \Rightarrow$$

$$\Rightarrow \mu(m) \le 2 \cdot (\mu(n) - m). \blacksquare$$

2.4.- *Lema.*

$$0 < m < \mu(n) \quad \Rightarrow \quad \mu(n - \mu(n) + m) \le 2 \cdot (\mu(n) - m)$$

Prueba:

Si probamos que $\mu(n - \mu(n) + m) = \mu(m)$ habremos probado el lema, en virtud del lema 2.3.

Sea $\mu(n) = F_k$.

Si $n = F_k$ (i.e. n es un número de Fibonacci), entonces ya está probado el lema.

Si $n > F_k$, entonces es claro que $\mu(n - F_k) = F_{k'}$, con $k' \ge (k + 2)$.

Por otra parte, al ser $m < F_k$, el mayor sumando que aparece en la descomposición de m será $F_{k''}$ con $k'' \le (k - 1)$.

Así, $n - F_k + m = \ldots + F_{k'} + F_{k''} + \ldots$ y no hay simplificación posible, de donde resulta claramente que $\mu(n - F_k + m) = \mu(m)$. $\blacksquare$

2.5.-*Lema.*

$$0 < m < \mu(n) \quad \Rightarrow \quad \mu(n - m) \le 2 \cdot m$$

Prueba:

En efecto: $0 < m < \mu(n) \quad \Rightarrow$

$\Rightarrow \quad 0 < (\mu(n) - m) < \mu(n)$; por el lema 2.4:

$\Rightarrow \quad \mu(n - \mu(n) + (\mu(n) - m)) \le 2 \cdot (\mu(n) - (\mu(n) - m)) \quad \Rightarrow$

$\Rightarrow \quad \mu(n - m) \le 2 \cdot m.$ $\blacksquare$

Representaremos los estados del juego por $\langle n,q \rangle$ donde

n = número de objetos en la pila

q = número máximo de objetos que podemos retirar.

Nótese que la situación inicial es $\langle n,n-1\rangle$.

2.6.- *Teorema.*

Para el juego descrito, consideremos la estrategia:
Dado $\langle n,q\rangle$ tomar n (si $n \le q$) ó $\mu(n)$ (en caso contrario).
Entonces: si en el estado inicial $\mu(n) \ne n$ (i.e. n no es un número de Fibonacci), esta estrategia es ganadora para el jugador que comienza el juego.

Prueba:

Dado $\langle n,q\rangle$ con $\mu(n) \le q$ se toma $\mu(n)$, y se pasa al estado: $\langle n-\mu(n), 2\cdot\mu(n)\rangle$.

El oponente no podrá ni ganar ni repetir esta estrategia, pues por el lema 2.2: $n - \mu(n) \ge \mu(n - \mu(n)) > 2\cdot\mu(n)$.

Así, el oponente encuentra un estado

$$\langle n,q\rangle \quad \text{con} \quad \mu(n) > q.$$

Supongamos que el oponente retira m objetos, con

$$0 < m \le q < \mu(n) .$$

El estado resultante es $\langle n-m, 2\cdot m\rangle$; por el lema 2.5, se tiene:

$$\mu(n - m) \le 2\cdot m$$

luego se puede ganar si $n - m \le 2\cdot m$, ó repetir la estrategia de nuevo. ■

2.7.- *Corolario.*

Para estados $\langle n,q\rangle$ con $n > q$ y $\mu(n) > q$ no es ganadora esta estrategia.

Prueba:

Si para un estado $\langle n,q\rangle$, se tiene $n > q$, significa que no podemos ganar en una jugada, pues no podemos retirar los n objetos que quedan en la pila. Además, si $\mu(n) > q$, tampoco podemos seguir la estrategia dada en el teorema 2.6. Esto quiere decir, que debido también a la estrategia del teorema 2.6 que garantiza la alternancia de su aplicación, el jugador contrario puede aplicar la estrategia a la que estamos aludiendo y en consecuencia, el jugador contrario gana el juego.

En este sentido hay que entender que la estrategia descrita no es ganadora para estados en las condiciones del enunciado del corolario. ■

§.3.- CONCLUSION.

Como se pone de manifiesto con el ejemplo estudiado, la demostración de existencia y el descubrimiento de estrategias ganadoras para juegos (incluso simples), son problemas muy difíciles. Por esta razón, se hace precisa la consideración de funciones heurísticas de evaluación de posiciones, usadas por procedimientos que eviten el desarrollo exhaustivo de los árboles de juego. A la formalización de estos procedimientos estará dedicado el Capítulo III.

CAPITULO I.- FORMALIZACION BASICA.

§.1.- PRELIMINARES.

§.2.- JUEGOS.

§.1.- PRELIMINARES.

1.1.- *Definición.*

Una *Relación Binaria* entre dos conjuntos A y B, es cualquier subconjunto $S \subseteq A \times B$.

Escribiremos $\langle a,b \rangle \in S$ ó aSb.

Notaremos: $S^{-1} = \{\langle b,a \rangle : aSb\} \subseteq B \times A$

$S(a) = \{x \in B : aSx\}$, para $a \in A$

$S^{-1}(b) = \{x \in A : xSb\}$, para $b \in B$

$S[C] = \{x \in B : \exists c \, (c \in C \wedge cSx)\}$, para $C \subseteq A$

$S^{-1}[C] = \{x \in A : \exists c \, (c \in C \wedge xSc)\}$, para $C \subseteq B$.

1.2.- *Definición.*

Un *Grafo* es un par $\langle N,S \rangle$ donde N es un conjunto (no vacío) y $S \subseteq N \times N$, tales que:

G.1.- La relación en N, que notamos por Ω, y que está definida por:

$$n \Omega m \leftrightarrow \exists k \geq 1 \,,\, \exists n_0, \ldots, n_k \in N$$
$$n_0 = n \wedge n_k = m \wedge [1 \leq i \leq k \rightarrow n_i \in S(n_{i-1})]$$

es un orden parcial.

G.2.- $\forall n \in N \; [S(n) \text{ finito}]$.

G.3.- $\exists n \in N \; [S^{-1}(n) = \emptyset]$, llamado nodo inicial ó raíz.

Los elementos de N se llaman *Nodos* del grafo; los elementos $\langle n,m \rangle \in S$ se llaman *Arcos* del grafo. Una sucesión de nodos, $\{n_0, n_1, \ldots\}$, se llama un *Camino* si $n_i S n_{i+1}$ $\forall i \geq 0$. Cuando un camino es finito, $\{n_0, n_1, \ldots, n_k\}$, diremos que su longitud (número de arcos) es k.

1.3.- *Notas.*

Esta no es la definición habitual de grafo; la hemos adoptado así por las razones que enseguida veremos.

Podíamos haber definido "grafo de juego" o algo por el estilo, pero por razones de comodidad, lo llamaremos simplemente grafo. Como es natural, nuestra intención será, más adelante, caracterizar los juegos mediante su representación con grafos; por este motivo, esta definición está adecuada al uso posterior.

La idea básica es simple: los nodos representan los estados del juego, y la relación S nos presenta los movimientos posibles.

La condición G.1 nos dice que no podemos, tras realizar uno ó más movimientos, volver al mismo estado en que nos encontramos; en particular nos dice que todo movimiento debe producir un cambio en el estado del juego.

La condición G.2 nos dice que, ante un estado del juego, sólo es posible un número finito de jugadas.

Si nSm (ó $m \in S(n)$) se dice que m es sucesor de n, y n antecesor de m; el conjunto de sucesores de n es pues $S(n)$ y el de antecesores $S^{-1}(n)$.

Finalmente, la condición G.3 nos indica que hay un estado privilegiado, sin antecesores, que corresponde al estado inicial del juego.

La representación gráfica que utilizamos para representar grafos se define:

- Los nodos se representan por puntos y/o etiquetas con la identificación del nodo.
- Los arcos se representan por segmentos que unen los nodos que determinan el arco. Si el sentido descendente no basta, usaremos una punta de flecha para indicar sucesión.

Como tenemos por la condición G.3, la existencia de un nodo raíz, será a partir de él desde donde comencemos a realizar la representación gráfica. Dibujamos los nodos sucesores de un nodo dado, que no hayan sido dibujados. Unimos el nodo dado con cada uno de sus sucesores mediante un arco. Repetimos el proceso para todos los nodos. Si un nodo no tiene sucesores, no dibujamos ningún arco.

1.4.- *Definición.*

Un grafo $\langle N,S\rangle$ se dice *Camino Finito* ó *C-finito*, si no admite caminos infinitos.

Un grafo $\langle N,S\rangle$ se dice *Camino Acotado* ó *C-acotado* si

$\forall n \in N \quad \exists K(n) \in \mathbb{N} \quad : \quad \forall$ camino $\{n, n_1, \ldots, n_k\}$ $k \leq K(n)$.

Nótese que si un grafo es C-acotado, entonces es C-finito.

1.5.- *Lema de König.*

Todo grafo C-finito es C-acotado.

Nota: La hipótesis de que el número de sucesores de cada nodo sea finito es esencial. En general, este resultado no es válido sin esa hipótesis. Con ella, C-finito y C-acotado son equivalentes.

Prueba:

Por reducción al absurdo: sea $\langle N,S\rangle$ un grafo C-finito, que no sea C-acotado. Existirá un nodo n tal que $\forall k \in \mathbb{N}$ hay un camino que comienza en n de longitud mayor que k.

Si para cada $n' \in S(n)$ existiese un número $K(n')$ tal que todos los caminos que comienzan en n' tuvieran longitud menor ó igual que $K(n')$, entonces, puesto que $S(n)$ es finito, existiría un entero maximal K' tal que todos los caminos que comienzan en n tendrían longitud menor que $K' + 1$.

Por consiguiente, debe existir al menos un nodo $n' \in S(n)$ tal que $\forall k \in \mathbb{N}$ existe un camino que comienza en n' de longitud mayor que k.

De forma análoga, existirá un $n'' \in S(n')$ con la misma propiedad, y así sucesivamente, podemos construir un camino infinito $\{n, n', n'', \ldots\}$ lo cual va en contra de la hipótesis de ser $\langle N,S\rangle$ C-finito. ■

1.6.- *Definición.*

Un subconjunto $K \subseteq N$ se llama un *Núcleo* del grafo $\langle N,S\rangle$, si cumple:

$$\forall n \in N \quad [n \in K \leftrightarrow S(n) \cap K = \emptyset]$$

1.7.- *Proposición.*

Si $\langle N,S \rangle$ es un grafo C-finito, entonces tiene uno y solo un núcleo.

Prueba:

P1.- Existencia.

Sea $K_0 = \{n \in N : S(n) = \emptyset\}$; $K_0 \neq \emptyset$ pues $\langle N,S \rangle$ es C-finito.

Para $i \geq 0$ pongamos $K'_i = \bigcup_{j=0}^{i} K_j$, y

$$K_{i+1} = \{n \in N : n \notin K'_i \wedge S(n) \cap K'_i = \emptyset \wedge$$
$$\forall m\ [m \in S(n) \Rightarrow S(m) \cap K'_i \neq \emptyset]\}$$

(ver figura 1.7)

Sea $K = \bigcup_{i \geq 0} K_i$.

Nótese que, si $n \in K_i$, entonces podemos encontrar un camino de longitud $2 \cdot i$ en el grafo; al ser éste C-finito, a partir de un índice los conjuntos K_i son vacíos; por tanto, K es la unión de un número finito de conjuntos. Veamos que K es el núcleo.

$\Longrightarrow]$ $n \in K \Rightarrow S(n) \cap K = \emptyset$:

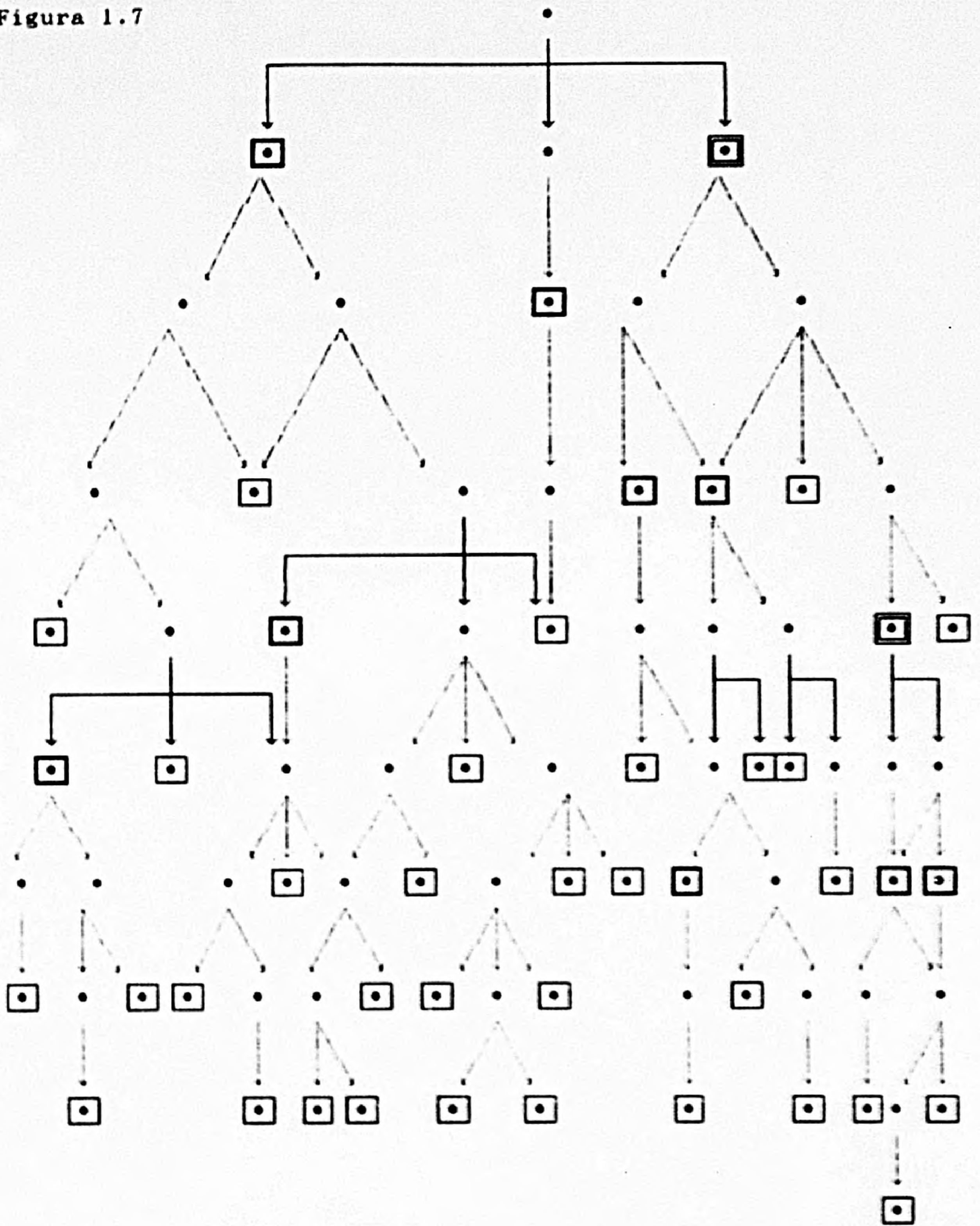
Si $n \in K_0$, entonces $S(n) = \emptyset$ y no hay nada que probar pues, trivialmente se tiene que $S(n) \cap K = \emptyset$.

Supongamos $n \in K_i$, $i \geq 1$.

Sea $m \in S(n) \cap K$; por la definición de los K_i , será $m \in K_j$ con $j \geq i$.

También por la definición de los K_i, $S(m) \cap K'_{i-1} \neq \emptyset$, luego $S(m) \cap K'_{j-1} \neq \emptyset$ ya que $K'_{i-1} \subseteq K'_{j-1}$; pero si $S(m) \cap K'_{j-1} \neq \emptyset$, entonces de nuevo por la definición de los K_i , $m \notin K_j$, lo que es contradictorio.

Figura 1.7



K_0 = nodos marcados con ⊡

K_1 = nodos marcados con ⊡

K_2 = nodos marcados con ⊡

$K_n = \emptyset$ para $n \geq 3$.

Veamos ahora la implicación contraria.

$\Longleftarrow$ $S(n) \cap K = \emptyset \Rightarrow n \in K$:

Si $n \notin K$, negando que $n \in K_{i+1}$ se tiene:

$$n \in K'_i \ \vee \ S(n) \cap K'_i \neq \emptyset \ \vee \ \exists m \in S(n)\ [S(m) \cap K'_i = \emptyset]$$

Ahora bien, tenemos: $n \in K'_i$ no puede darse pues $n \notin K$.

$S(n) \cap K'_i \neq \emptyset$ tampoco puede darse pues $S(n) \cap K = \emptyset$, luego $\exists m \in S(n)\ [S(m) \cap K'_i = \emptyset]$; puesto que los $K'_i \neq \emptyset$ son en número finito, tomando el de menor índice que sea vacío, resulta:

$$\exists m \in S(n) \ [S(m) \cap K = \emptyset].$$

Por tanto, si $n \notin K \wedge S(n) \cap K = \emptyset$, hemos encontrado un

$$m \in S(n) \ \text{ tal que } \ m \notin K \wedge S(m) \cap K = \emptyset.$$

Este proceso, repetido indefinidamente, nos daría un camino infinito, lo cual resulta contradictorio.

P2.- Unicidad.

Sea K^* otro núcleo. Veamos que $K = K^*$ por doble inclusión.

Veamos que $K \subseteq K^*$.

Es claro que $K_0 \subseteq K^*$. Veamos por inducción, que $K_{i+1} \subseteq K^*$.

Sea $n \in K_{i+1}$. Si $n \notin K^*$, será $S(n) \cap K^* \neq \emptyset$; tomemos

$$m \in S(n) \cap K^*.$$

Tenemos $m \in S(n) \ \wedge \ n \in K_{i+1} \ \Rightarrow \ S(m) \cap K'_i \neq \emptyset$, y como por hipótesis de inducción $K'_i \subseteq K^*$ resulta $S(m) \cap K^* \neq \emptyset$, de donde $m \notin K^*$ (definición de núcleo), lo que es contradictorio con la elección de m. Así, $K \subseteq K^*$.

Veamos el recíproco $K^* \subseteq K$.

Sea $n \in K^*$; será $S(n) \cap K^* = \emptyset$; pero como $K \subseteq K^*$ se tendrá $S(n) \cap K = \emptyset$, de donde, por ser K núcleo, $n \in K$. ■

§.2.- JUEGOS.

Formalización y estudio de juegos para dos jugadores.

2.1.- *Definición.*

Un *Juego* es una 5-tupla $\langle E,R,T,G,P \rangle$, donde

E es un conjunto finito no vacío

T, G y P son subconjuntos de E

$R \subseteq E \times E$

tales que:

J.1.- $(G \cup P) \cap R^{-1}[E] = \emptyset$

J.2.- $P \subseteq T$

J.3.- $G \cap T = \emptyset$

J.4.- $\forall s,t \in E \quad [sRt \Rightarrow (s \in T \Leftrightarrow t \notin T)]$

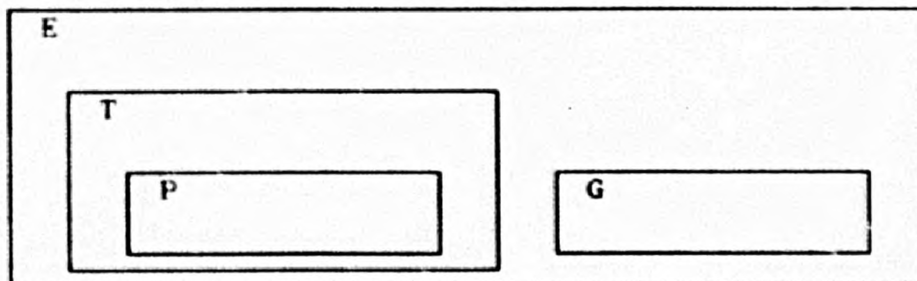
J.5.- $\langle E,R \rangle$ es un grafo.

2.2.- *Notas.*

Pretendemos caracterizar, de la manera más general posible, los juegos de dos jugadores que juegan alternativamente. La situación será que "nosotros" jugamos contra un "contrincante" y tratamos de encontrar estrategias que nos resulten favorables.

El conjunto E es el conjunto de los estados posibles del juego.

El subconjunto T caracteriza las situaciones o estados en que nos toca mover a nosotros. Los elementos de G son los estados ganadores; si al mover, alcanzamos uno de estos estados, hemos ganado el juego.



Finalmente, P es el conjunto de estados perdedores; si al mover el contrincante, alcanza uno de estos estados, entonces hemos perdido el juego (esto es, el contrincante ha ganado).

Nótese que, en principio, no excluimos la posibilidad de empate.

La relación R establece las reglas del juego: sRt quiere decir que del estado s es posible, con un movimiento permitido, pasar al estado t.

Comentaremos algo más la definición. $R^{-1}[E]$ es el conjunto de los nodos que tienen sucesores, i.e.

$$R^{-1}[E] = \{s \in E : R(s) \neq \emptyset\}$$

Así, J.1 de la definición 2.1 puede leerse:

$$G \cup P \subseteq (E - R^{-1}[E])$$

que quiere decir que al alcanzarse un estado perdedor ó ganador, el juego ha terminado.

Nótese que $E - R^{-1}[E]$ es el conjunto de los nodos que no tienen sucesores, así que, por ejemplo, la posibilidad de empate se elimina exigiendo que $G \cup P = (E - R^{-1}[E])$

ó lo que es equivalente $E = G \cup P \cup R^{-1}[E]$.

Supondremos también que nosotros nunca haremos un movimiento que produzca un estado perdedor; esto lo escribimos

$(E - T) \cap P = \emptyset$ que a su vez es equivalente a

$P \subseteq T$ como establece J.2 de la definición 2.1.

Análogamente, el contrincante no hará un movimiento que produzca un estado ganador, esto es,

$T \cap G = \emptyset$, como establece J.3 de la definición 2.1.

Estas condiciones son razonables y no son restrictivas; puede objetarse que hay juegos en los que uno está obligado a hacer un movimiento que conduzca a la victoria del oponente, lo que no encaja con las condiciones que estamos exigiendo; esta disconformidad es solo aparente, y se corrige sin más que modificar los conjuntos G y P : si por ejemplo, estamos obligados a realizar un movimiento que conduzca a nuestra derrota, entonces el estado de partida ya es un estado perdedor.

Por su parte la condición J.4 de la definición 2.1, no hace más que expresar la alternancia del juego, mientras que J.5 quedó comentada tras la definición 1.2 de grafo.

2.3.- *Ejemplo.*

Para el caso del juego descrito inicialmente en el capítulo anterior, cada estado puede venir dado por una terna $\langle n,q,i\rangle$ donde:

n = número de objetos de la pila
q = el tope de piezas que pueden retirarse
i = actúa como indicador: si es 1 nos toca mover, y si es 0 le toca al contrincante.

Así, podemos poner, para una situación inicial con 15 objetos:

$E = \{\langle n,q,i\rangle : 0 \le n \le 15,\ 2 \le q \le 28,\ 0 \le i \le 1\}$
$T = \{\langle n,q,1\rangle \in E\}$
$G = \{\langle 0,q,0\rangle \in E\}$
$P = \{\langle 0,q,1\rangle \in E\}$
$R = \{\langle\langle n,q,i\rangle,\langle n',2\cdot(n-n'),1-i\rangle\rangle \in E{\times}E : 0 < n-n' \le q\}$

Nótese que la descripción de E, por mor de ser cómoda, contiene estados superfluos; ya vimos en la figura 0-2.1, al desarrollar el grafo del juego, que supondríamos que ningún jugador haría, si le era posible, una jugada que condujera a la victoria inmediata del adversario. Así, el estado $\langle 1,28,0\rangle$ no va a producirse tras el inicial $\langle 15,14,1\rangle$; por otra parte, el estado $\langle 15,2,1\rangle$ no se dará nunca,... pero, como hemos dicho, la descripción precisa de E es muy complicada, y no nos proporciona ninguna ventaja que merezca tal precisión.

2.4.- *Definición.*

Dado un juego (si no es preciso, no detallaremos $\langle E,R,T,G,P\rangle$) una *Estrategia* Q es una función parcial,

$$Q: T \longrightarrow E. \text{ tal que:}$$

E.1.- $Q(s) = t \Rightarrow sRt$

E.2.- $[Q(s) = t \wedge tRu \wedge R(u) \neq \emptyset] \Rightarrow \exists v \in E\ [Q(u) = v]$

2.5.- *Notas.*

Una estrategia es una función que determinará nuestros movimientos; la condición E.1 nos dice que Q sólo hace movimientos legales, y la condición E.2 nos asegura que Q está definida para cualquier estado que se produzca como consecuencia de movimientos del contrincante.

Si $[Q(s) = t \wedge t \in G]$, entonces ganamos el juego. Por contra, si $\exists u \in P\ [Q(s) = t \wedge tRu]$, entonces podemos perder al siguiente movimiento.

2.6.- *Ejemplo.*

Para el ejemplo que venimos considerando, tenemos la estrategia que consiste en retirar 1 objeto cada vez:

$$Q': T \longrightarrow E \qquad Q'(\langle n,q,1\rangle) = \langle n-1,2,0\rangle \quad \text{si} \quad n \geq 1.$$

Esta función es parcial pues no está definida por ejemplo para los $\langle n,q,0\rangle$ ni para los $\langle 0,q,i\rangle$.

Ya hemos visto que para la situación inicial $\langle 15,14,1\rangle$ ganamos el juego con la estrategia:

$$Q: T \longrightarrow E \qquad Q(\langle n,q,1\rangle) = \begin{cases} \langle 0,2\cdot n,0\rangle & \text{si} \quad n \leq q \\ \langle n-\mu(n),2\cdot\mu(n),0\rangle & \text{si} \quad n > q \end{cases}$$

2.7.- *Definición.*

Dado un juego, una estrategia Q, y un estado $s_1 \in T$, una *Q-sucesión* para s_1 es una sucesión de estados $\{s_1,s_2,\ldots,s_n\}$ tal que:

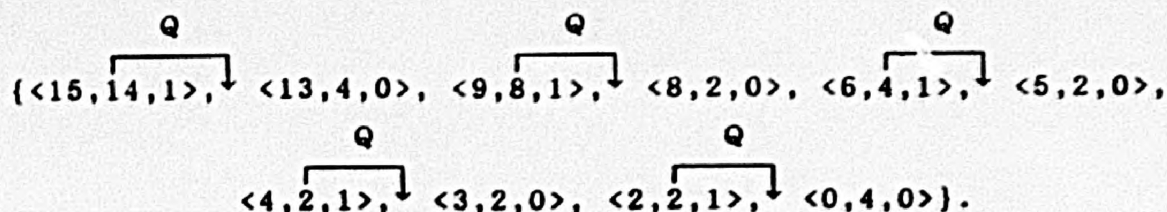
i) $\forall i \ [1 \leq i \leq n-1 \Rightarrow s_i R s_{i+1}]$

ii) $\forall i \ [1 \leq i \leq n-1 \wedge i \text{ impar} \Rightarrow Q(s_i) = s_{i+1}]$

iii) $R(s_n) = \emptyset$.

2.8.- *Ejemplo.*

Una Q-sucesión en el juego que venimos considerando es, como puede verse en la figura 0-2.1,



2.9.- *Definición.*

Dado un juego, una estrategia Q, y un estado $s \in T$, diremos que Q es una *Estrategia Ganadora para* s, si toda Q-sucesión para s termina con un estado de G.

2.10.- *Ejemplo.*

En el grafo de la figura 0-2.1 puede comprobarse que la estrategia Q descrita en 2.6 es efectivamente ganadora para <15,14,1>, cosa que hemos demostrado en el capítulo anterior.

2.11.- *Proposición.*

Dado un juego, una estrategia Q y un estado $s_1 \in T$, si Q es ganadora para s_1, entonces toda Q-sucesión para s_1 es de longitud impar.

Prueba:

Sea $\{s_1, s_2, \ldots, s_n\}$ una Q-sucesión para s_1.

De las definiciones se sigue que $s_i \in T$ para i impar. Si $s_n \in G$ es el último elemento de la Q-sucesión, será $s_n \notin T$ (por J.3 de la definición 2.1), luego n es par. Por tanto, la longitud, que es $n - 1$, será impar. ■

2.12.- *Definición.*

Una *Evaluación* de un juego es una colección $\{V_i\}_{i \geq 1}$ de subconjuntos de E, definidos por:

$$V_1 = \{s \in E : R(s) \cap G \neq \emptyset\}$$

para $i \geq 1$, ponemos $V'_i = \bigcup_{j=1}^{i} V_j$ y definimos:

$$V_{i+1} = \{s \in E : s \notin V'_i \wedge \exists t \in R(s) \; [\emptyset \neq R(t) \subseteq V'_i]\}$$

2.13.- *Notas.*

Esta definición pretende caracterizar los estados que nos resultan favorables. Es claro que si $s \in V_1$, y nos toca mover a nosotros (ver proposición 2.15), entonces con un solo movimiento ganamos el juego.

Si por ejemplo, $s \in V_2$, y de nuevo nos toca mover a nosotros, haríamos el movimiento que consiste en pasar al estado t de la definición desde el que, haga lo que haga el contrincante, se pasa a V_1, desde donde ganamos el juego.

2.14.- *Ejemplo.*

Para el ejemplo que venimos considerando es claro que

$$V_1 = \{\langle n,q,1\rangle : n \le q\}.$$

Sin embargo, suponiendo que el contrincante hace sus jugadas evitando que ganemos a la siguiente jugada, los estados, tomados del grafo de la figura 0-2.1 son:

$$V_1 = \{\langle 2,2,1\rangle\}$$

$$V_2 = \{\langle 4,2,1\rangle\}$$

$$V_3 = \{\langle 7,2,1\rangle,\ \langle 6,4,1\rangle\}$$

$$V_4 = \{\langle 10,2,1\rangle,\ \langle 9,4,1\rangle,\ \langle 11,4,1\rangle,\ \langle 10,6,1,\rangle,\ \langle 9,8,1\rangle\}$$

$$V_5 = \{\langle 12,2,1\rangle\}$$

$$V_6 = \{\langle 15,14,1\rangle\}$$

$$V_k = \emptyset \quad \text{para} \quad k \ge 7.$$

2.15.- *Proposición.*

En las condiciones de la definición 2.12, $\forall i\ [V_i \subseteq T]$.

Prueba:
Por inducción.

Si $s \in V_1$, entonces $\exists t \in R(s) \cap G$;

por J.3 de la definición 2.1, $t \in G \rightarrow t \notin T$

por J.4 de la misma definición $t \notin T \wedge t \in R(s) \Rightarrow s \in T$

luego $V_1 \subseteq T$.

Sea ahora $s \in V_{i+1}$. Por la definición 2.12, existirán

$t,u \in E$ tales que: $sRt \wedge tRu \wedge u \in V'_i$.

Por la hipótesis de inducción $u \in T$, luego al haber hecho dos movimientos, por J.4 de la definición 2.1, tenemos que $s \in T$.■

2.16.- *Definición.*

Con las notaciones anteriores, una estrategia se dice *Evaluadora* si se tiene:

i) $s \in V_1 \Rightarrow Q(s) \in G$

ii) $s \in V_{i+1} \Rightarrow \emptyset \neq R(Q(s)) \subseteq V'_i$

2.17.- *Nota.*

La idea es clara: Q debe conducir directamente a la victoria, o permitir, sea cual sea el movimiento del contrincante, colocarnos de nuevo en una posición favorable. Como es de esperar se tiene:

2.18.- *Proposición.*

Una estrategia evaluadora Q es ganadora para cualquier estado de $V = \bigcup_{i \geq 1} V_i$. V se llama *Conjunto Evaluador* del juego.

Prueba:

Sea $s \in V$; será $s \in V_k$ para cierto $k \in \mathbb{N}$. Por inducción sobre k probaremos que Q es ganadora para s.

Si $k = 1$, entonces $Q(s) \in G$, por definición de estrategia evaluadora; así, $(s,Q(s))$ es la única Q-sucesión para s.

Sea ahora $s \in V_{k+1}$, $k > 0$. Será $\emptyset \neq R(Q(s)) \subseteq V'_k$; sea $s' \in R(Q(s))$, y sea $(s,Q(s),s',\ldots)$ una Q-sucesión para s ; es claro que $(s',\ldots)$ es una Q-sucesión para s' ; por hipótesis de inducción Q es ganadora para s', luego es ganadora para s también. ■

2.19.- *Definición.*

Con las notaciones anteriores, una estrategia se dice *Cauta* si se tiene:

$$\forall s \in V \quad [Q(s) \in G \quad \vee \quad \emptyset \neq R(Q(s)) \subseteq V]$$

2.20.- *Notas.*

Comparación entre estrategias Cautas y Evaluadoras.

Las estrategias Evaluadoras nos hacen pasar, de un estado perteneciente a un determinado conjunto V_k, a un estado perteneciente a un conjunto $V_{k'}$, con $k' < k$, i.e. nos acercan a la solución. Las estrategias Cautas sólo nos mantienen en posiciones favorables.

Es claro que si Q es una estrategia evaluadora, entonces es cauta, pero no al contrario.

2.21.- *Proposición.*

Si el grafo $\langle E,R \rangle$ del juego $\langle E,R,T,G,P \rangle$ es un grafo C-finito, entonces una estrategia cauta es ganadora para cualquier estado de V, siendo V el conjunto evaluador del juego.

Nótese que, al exigir en la definición 2.1 que E sea finito, el grafo $\langle E,R \rangle$ es siempre c-finito. En efecto, la única posibilidad de caminos infinitos consistiría en admitir repeticiones de nodos, lo que está excluído por G.1 de la definición 1.2.

Prueba:

Sea $s_1 \in V$; por ser $\langle E,R \rangle$ C-finito, cualquier Q-sucesión para s_1 es finita. Sea $(s_1, s_2, \ldots, s_n)$ una Q-sucesión para s_1. Hemos de probar que $s_n \in G$.

Es claro que para los i impares, es $s_i \in V$: en efecto, para $i = 1$ es cierto por hipótesis; y en general, por la definición de estrategia cauta:

$$s_i \in V \rightarrow [Q(s_i) \in G \quad \vee \quad \emptyset \neq R(Q(s_i)) \subseteq V] ,$$

esto es, $s_i \in V \rightarrow [s_{i+1} \in G \quad \vee \quad \emptyset \neq R(s_{i+1}) \subseteq V]$, con lo que $s_{i+2} \in V$.

Sea ahora i impar de nuevo: Si $Q(s_i) \in G$, la sucesión termina con $s_n = s_{i+1} \in G$, como se desea.

Si $s_{i+1} \notin G$, entonces como $s_{i+2} \in R(s_{i+1}) = R(Q(s_i)) \subseteq V$, repetimos el proceso con s_{i+2}. Este proceso es claramente finito. ■

2.22.- *Proposición.*

Con las notaciones habituales:

$$\forall s \in T \quad [\exists t \in R(s) \quad [\emptyset \neq R(t) \subseteq V] \Rightarrow s \in V].$$

Prueba:

Sean s,t como en la hipótesis.

Como $R(t) \subseteq V$, para cada $s' \in R(t)$ existirá un $i_{s'}$, tal que $s' \in V_{i_{s'}}$; poniendo $i_0 = \max\{i_{s'}: s' \in R(t)\}$, se tendrá $R(t) \subseteq V'_{i_0}$. Si $s \in V'_{i_0}$, entonces $s \in V$ como se desea. Si no, por definición de V_i será $s \in V_{i_0+1}$, luego de nuevo $s \in V$. ■

2.23.- *Proposición.*

Con las notaciones habituales:

Si existe una estrategia ganadora Q para un estado $s \in T$, entonces $s \in V$.

Prueba:

Sea $s \in T$; sea $n(s)$ la longitud de una Q-sucesión para s de longitud máxima.

Como $s \in T$ será $s \notin G$; por tanto $Q(s)$ debe estar definido pues si no, (s) sería una Q-sucesión que no termina en G. Así, $n(s) \geq 1$.

Razonamos por inducción sobre $n(s)$.

Si $n(s) = 1$, entonces $Q(s) \in G$ y por tanto $s \in V_1 \subseteq V$.

Sea ahora $n(s) > 1$. Desde luego $n(s)$ es impar, pues por la proposición 2.11, todas las Q-sucesiones son de longitud impar. Es claro que $\forall s' \in R(Q(s))$ será $n(s') < n(s)$, y, por hipótesis de inducción $s' \in V_k$ para algún k. Dado que $R(Q(s))$ es finito, si k_0 es el máximo de tales k, se tiene: $R(Q(s)) \subseteq V'_{k_0}$. Luego

$$s \in V'_{k_0+1} \subseteq V. \text{ ■}$$

2.24.- *Proposición.*

Sea un juego tal que:

$$E = R^{-1}[E] \cup G \cup P.$$

Sea K el núcleo de <E,R>. Se tiene:

$$\forall s \in E \quad [s \in V \iff R(s) \cap (K - T) \neq \emptyset]$$

Prueba:

⟹| Condición necesaria.

Sea $s \in V$; será $s \in V_i$ para algún i. Por inducción sobre i.

Si $i = 1$, entonces podemos tomar $t \in R(s) \cap G$; por J.3 de la definición 2.1, se tiene que $t \notin T$, y $t \in K$ (pues $t \in G \Rightarrow R(t) = \emptyset \Rightarrow R(t) \cap K = \emptyset$), luego $t \in R(s) \cap (K - T)$ y $R(s) \cap (K - T) \neq \emptyset$.

Sea ahora $s \in V_{i+1}$, $i > 0$.

Podemos tomar ahora $t \in R(s)$ con $R(t) \subseteq V'_i$; por hipótesis de inducción $\forall s' \in V'_i$ $R(s') \cap (K - T) \neq \emptyset$; en particular, $R(s') \cap K \neq \emptyset$, luego $s' \notin K$ (definición de núcleo). Al ser esto válido $\forall s' \in V'_i$, y ser $R(t) \subseteq V'_i$, se tendrá: $R(t) \cap K = \emptyset$, de donde, de nuevo por la definición de núcleo, $t \in K$.

Como antes, $[s \in T \wedge t \in R(s)] \Rightarrow t \notin T$, luego:

$$t \in R(s) \cap (K - T), \text{ esto es, } R(s) \cap (K - T) \neq \emptyset$$

⟸| Condición suficiente.

Sea $t \in R(s) \cap (K - T)$.

Si $R(t) = \emptyset$ entonces, del enunciado se deduce que $t \in G \cup P$; como $P \subseteq T$ (por J.2 de la definición 2.1), será $t \in G$, y así, $s \in V_1 \subseteq V$.

Supongamos ahora que $R(t) \neq \emptyset$.
Sea Q una estrategia definida sobre T de cualquier modo, con tal que se tenga $Q(u) \in K$ cuando $u \notin K$ (esto es posible por la definición de núcleo).

Sea $s_1 \in R(t)$, y consideramos $\{s_1, s_2, \ldots, s_n\}$ una Q-sucesión para s_1 ($\langle E,R\rangle$ es C-finito).

Como $s_n \notin R^{-1}[E]$, se tiene que $s_n \in G \cup P$.

Si $s_n \in P$, será $s_n \in T$; como $s_1 \in T$ (pues $t \notin T$) será n impar. Una sencilla inducción, dada la elección de Q, nos muestra que $s_i \notin K$ para los i impares, luego $s_n \notin K$. Pero esto es contradictorio con que $R(s_n) = \emptyset$, pues esto implica que $s_n \in K$. Por consiguiente, $s_n \in G$, y por tanto Q es una estrategia ganadora para s_1. Por la proposición 2.23, $s_1 \in V_i$.

Así hemos probado que $R(t) \subseteq V$; aplicando la proposición 2.22, resulta que $s \in V$. ■

2.25.- *Definición.*

Se dice que un juego $\langle E,R,T,G,P\rangle$ es *Interpretable en Grafos* si existe un grafo $\langle N,R'\rangle$, que se llamará *Grafo del Juego*, y un subconjunto $F \subseteq N$, tales que:

i) $E = N \times \{0,1\}$

ii) 1) $G = \{\langle n,0\rangle : n \in F\}$

2) $P = \{\langle n,1\rangle : n \in F\}$

iii) 1) $n_1 R' n_2 \leftrightarrow \langle n_1,0\rangle R \langle n_2,1\rangle$

2) $n_1 R' n_2 \leftrightarrow \langle n_1,1\rangle R \langle n_2,0\rangle$

iv) $T = \{\langle n,1\rangle : n \in N\}$

2.26.- *Notas.*

Es claro que N, R', F determinan completamente un juego interpretable en grafos.

Un juego interpretable en grafos es un juego simétrico, esto es, las reglas del juego no favorecen a ninguno de los jugadores.

Precisamente esta simetría produce un cierto conflicto en las definiciones: Si $\langle N,R'\rangle$ es un grafo, entonces tiene un único nodo inicial, n. La definición anterior nos lleva a que en el

grafo $\langle E,R\rangle$ existan dos nodos iniciales: $\langle n,0\rangle$ y $\langle n,1\rangle$, contra la definición de grafo que dimos en 1.2.

Creemos que no es preciso modificar las definiciones porque la idea de simetría implicará necesariamente esta dualidad. Por ejemplo, la situación inicial del juego del ajedrez es única; sólo es una convención el hecho de que comience la partida el jugador que juega con las piezas blancas, pero es evidente que el juego puede desarrollarse igualmente si comienza la partida el jugador que juega con las fichas negras.

2.27.- *Ejemplo.*

Para el juego descrito en el capítulo 0, se tiene:

$N = \{\langle n,q\rangle : 0 \le n \le 15 \ , \ 2 \le q \le 28\}$

$R' = \{\langle\langle n,q\rangle,\langle n',2\cdot(n-n')\rangle\rangle \in N{\times}N : 0 < n-n' \le q\}$

$F = \{\langle 0,q\rangle \in N\}$

2.28. - *Proposición.*

Sea $\langle E,R,T,G,P\rangle$ un juego interpretable en grafos, con grafo $\langle N,R'\rangle$ y subconjunto F. Se tiene:

i) $R'[F] = \emptyset$

ii) Si K es un núcleo para $\langle N,R'\rangle$, entonces $K' = K{\times}\{0,1\}$ es un núcleo para $\langle E,R\rangle$.

Prueba:

i) Supongamos $R'[F] \ne \emptyset$, y sea $n' \in R'[F]$; existirá un $n \in F$ tal que $nR'n'$;
por 2.25 iii): $\langle n,1\rangle R\langle n',0\rangle \ \wedge \ \langle n,0\rangle R\langle n',1\rangle$
pero por 2.25 ii): $\langle n,0\rangle \in G \ \wedge \ \langle n,1\rangle \in P$.
Estas conclusiones son contradictorias con el hecho de ser $R[G \cup P] = \emptyset$ (consecuencia de J.1 de la definición 2.1).

ii) Hay que demostrar que $s \in K' \iff R(s) \cap K' = \emptyset$.

$\Longrightarrow$ Condición Necesaria.

Sea $s \in K'$. Será $s = \langle n,0\rangle$ ó $s = \langle n,1\rangle$ con $n \in K$.
Sea pues, $s = \langle n,i\rangle$ $i = 0,1$; veamos que $R(s) \cap K' = \emptyset$. Si no lo fuera, existiría un $\langle n',1-i\rangle$ con $n' \in K$, tal que $\langle n,i\rangle R\langle n',1-i\rangle$; de aquí, $nR'n'$, con lo que $R'(n) \cap K \ne \emptyset$, lo que es contradictorio con ser K núcleo.

$\Longleftarrow$ Condición Suficiente.

Sea $s = \langle n,i \rangle$ tal que $R(s) \cap K' = \emptyset$.

Si no fuese $s \in K'$, sería $n \notin K$, de donde, por ser K núcleo, $R'(n) \cap K \neq \emptyset$.

Sea entonces $n' \in R'(n) \cap K$, y $s' = \langle n', 1-i \rangle$. Es claro que $s' \in R(s) \cap K'$ lo que contradice la hipótesis. ■

2.29.- *Definición.*

Dados dos juegos $\langle E,R,T,G,P \rangle$, y $\langle E',R',T',G',P' \rangle$, una aplicación $\Phi: E \longrightarrow E'$ se llama un *Isomorfismo de Juegos*, y éstos se dicen *Isomorfos* , si se tiene:

i) Φ es biyectiva

ii) $s_1 R s_2 \leftrightarrow \Phi(s_1) R' \Phi(s_2)$

iii) $\Phi[T] = T'$

iv) $\Phi[G] = G'$

v) $\Phi[P] = P'$.

2.30.- *Proposición.*

Si un juego $\langle E,R,T,G,P \rangle$ es interpretable en grafos, con grafo $\langle N,R' \rangle$ y subconjunto $F \subseteq N$, entonces existe una aplicación

$$\alpha: T \longrightarrow E - T \quad \text{tal que:}$$

i) es biyectiva

ii) $s_1 R s_2 \leftrightarrow [\alpha(s_1) R \alpha^{-1}(s_2) \ \vee \ \alpha^{-1}(s_1) R \alpha(s_2)]$

iii) $\alpha[P] = G$

Prueba:

Definamos $\alpha(\langle n,1 \rangle) = \langle n,0 \rangle$.

i) Es evidente que α así definida es biyectiva.

ii) Demostremos la doble implicación:

$\Longrightarrow$ Sean s_1, s_2 tales que $s_1 R s_2$. Será $s_1 = \langle n_1, 0 \rangle$ ó $s_1 = \langle n_1, 1 \rangle$ con $n_1 \in N$. Si $s_1 = \langle n_1, 0 \rangle$, entonces será, $s_2 = \langle n_2, 1 \rangle$, con $n_2 \in N$

Por iii) de la definición 2.25 se tiene:

$$\langle n_1,0 \rangle R \langle n_2,1 \rangle \rightarrow n_1 R' n_2 \rightarrow \langle n_1,1 \rangle R \langle n_2,0 \rangle \rightarrow \alpha^{-1}(s_1) R \alpha(s_2)$$

De forma análoga se comprueba que si $s_1 = \langle n_1, 1\rangle$, entonces

$$\alpha(s_1)R\alpha^{-1}(s_2).$$

$\Longleftarrow$ Si $\alpha(s_1)R\alpha^{-1}(s_2)$, entonces será $s_1 = \langle n_1, 1\rangle$, $s_2 = \langle n_2, 0\rangle$, con $n_1, n_2 \in N$.

De nuevo por iii) de la definición 2.25:

$\langle n_1, 0\rangle R\langle n_2, 1\rangle \Rightarrow n_1 R' n_2 \Rightarrow \langle n_1, 1\rangle R\langle n_2, 0\rangle \Rightarrow s_1 R s_2$.

De forma análoga se comprueba que si $\alpha^{-1}(s_1)R\alpha(s_2)$, entonces $s_1 R s_2$.

iii) $s \in \alpha[P] \Leftrightarrow s = \alpha(\langle n, 1\rangle),\ n \in F$
$\Leftrightarrow s = \langle n, 0\rangle,\ n \in F$
$\Leftrightarrow s \in G.$ ■

2.31.- *Nota.*

Nótese que el apartado ii) de la proposición 2.30, hace referencia a la simetría de que hablábamos antes. En efecto, de dicho apartado se deduce que si $\langle n, i\rangle R\langle n', 1-i\rangle$, entonces también $\langle n, 1-i\rangle R\langle n', i\rangle$, esto es, que cualquier jugada que sea válida para un jugador, lo es también para el otro.

2.32.- *Proposición.*

Sea $J = \langle E, R, T, G, P\rangle$ un juego. Si existe una aplicación

$$\alpha: T \longrightarrow E - T$$

verificando las condiciones i), ii) y iii) de la proposición 2.30, entonces J es isomorfo a un juego interpretable en grafos.

Prueba:

Construimos un grafo $\langle N, S\rangle$ tomando $N = T$, y definiendo

$$n_1 S n_2 \Leftrightarrow n_1 R\alpha(n_2) \quad ;$$

consideramos el conjunto $F = P$.

Construimos ahora un juego $J' = \langle E', R', T', G', P'\rangle$, tomando:

$$E' = N \times \{0,1\}$$

$$\langle n_1,0\rangle R'\langle n_2,1\rangle \leftrightarrow n_1 S n_2$$

$$\langle n_1,1\rangle R'\langle n_2,0\rangle \leftrightarrow n_1 S n_2$$

$$T' = \{\langle n,1\rangle : n \in N\}$$

$$G' = \{\langle n,0\rangle : n \in F\}$$

$$P' = \{\langle n,1\rangle : n \in F\}$$

Por la construcción, el juego J' es interpretable en grafos, con grafo $\langle N,S\rangle$ y subconjunto $F \subseteq N$. Nos bastará ver que J y J' son isomorfos. Definimos

$\Phi: E \longrightarrow E'$ por:

$$\Phi(s) = \begin{cases} \langle s,1\rangle & \text{si } s \in T \\ \langle \alpha^{-1}(s),0\rangle & \text{si } s \notin T \end{cases}$$

Veamos que Φ cumple las cinco condiciones de la definición 2.29.

i) Φ inyectiva puesto que:

$\Phi(s) = \Phi(s') \Rightarrow [s,s' \in T \wedge \langle s,1\rangle = \langle s',1\rangle] \vee$

$[s,s' \notin T \wedge \langle \alpha^{-1}(s),0\rangle = \langle \alpha^{-1}(s'),0\rangle] \Rightarrow$

$[s = s' \vee \alpha^{-1}(s) = \alpha^{-1}(s')] \Rightarrow s = s'$.

Φ sobreyectiva puesto que:

$s' \in E' \Rightarrow \exists s \in N$ (notar $N = T$): $[s' = \langle s,0\rangle \vee s' = \langle s,1\rangle]$. Si $s' = \langle s,0\rangle$, entonces $s' = \Phi(\alpha(s))$, y si $s' = \langle s,1\rangle$, entonces $s' = \Phi(s)$.

ii) Demostramos la doble implicación.

$\Longrightarrow$ Sea $s_1 R s_2$; será $s_1 \in T$ ó $s_1 \notin T$.

Si $s_1 \in T$, entonces $s_2 \notin T$ (por J.4 de la definición 2.1), luego $\Phi(s_1) = \langle s_1,1\rangle$, $\Phi(s_2) = \langle \alpha^{-1}(s_2),0\rangle$.

Por la construcción de S: $s_1 R s_2 \Rightarrow s_1 S \alpha^{-1}(s_2)$.

Por la construcción de R':

$$s_1 S \alpha^{-1}(s_2) \Rightarrow \langle s_1,1\rangle R' \langle \alpha^{-1}(s_2),0\rangle \Rightarrow \Phi(s_1) R' \Phi(s_2).$$

Si $s_1 \notin T$, entonces $s_2 \in T$ (por J.4 de definición 2.1), luego: $\Phi(s_1) = \langle \alpha^{-1}(s_1),0\rangle$, $\Phi(s_2) = \langle s_2,1\rangle$.

De la condición ii) de la proposición 2.30:

$$s_1Rs_2 \Rightarrow \alpha^{-1}(s_1)R\alpha(s_2) \vee \alpha(s_1)R\alpha^{-1}(s_2) \text{ y además,}$$

debe de ocurrir la primera de ambas, pues la segunda no puede darse al no estar definido $\alpha(s_1)$.

Ahora, por la construcción de S:

$$\alpha^{-1}(s_1)R\alpha(s_2) \Rightarrow \alpha^{-1}(s_1)Ss_2$$

y por la construcción de R':

$$\alpha^{-1}(s_1)Ss_2 \Rightarrow \langle \alpha^{-1}(s_1),0\rangle R'\langle s_2,1\rangle \Rightarrow \Phi(s_1)R'\Phi(s_2)$$

$\Longleftarrow$ Sea ahora $\Phi(s_1)R'\Phi(s_2)$. Será $s_1 \in T$ ó $s_1 \notin T$.
Si $s_1 \in T$, será $\Phi(s_1) = \langle s_1,1\rangle$; por la construcción de R' será $\Phi(s_2) = \langle \alpha^{-1}(s_2),0\rangle$ y además $\langle s_1,1\rangle R'\langle \alpha^{-1}(s_2),0\rangle \Rightarrow s_1S\alpha^{-1}(s_2)$ y por la construcción de S: $s_1S\alpha^{-1}(s_2) \Rightarrow s_1Rs_2$.

Si $s_1 \notin T$, la prueba es análoga.

iii) $\Phi[T] = \{\langle s,1\rangle : s \in T\} = T'$ por la construcción de T' y por ser $N = T$.

iv) $\Phi[G] = \{\langle \alpha^{-1}(s),0\rangle : s \in G\} =$
por J.3 de la definición 2.1, $= \{\langle s,0\rangle : s \in P\} =$
por iii) de la proposición 2.30, $= G'$ por la construcción de G' y por ser $F = P$.

v) $\Phi[P] = \{\langle s,1\rangle : s \in P\} =$ por J.1 de la definición 2.1,
$\Phi[P] = P'$ por la construcción de P' y por ser $F = P$. ■

2.33.- *Teorema.*

Sea $\langle E,R,T,G,P\rangle$ un juego, y sea K el núcleo de $\langle E,R\rangle$. Se tiene:

$\forall s\ [s \in K \cap T \Rightarrow$ no existe estrategia ganadora para $s]$.

Prueba:

Sea Q una estrategia cualquiera, y sea $s \in K \cap T$.
Si $R(s) = \emptyset$ (en particular si $s \in P$), entonces no hay nada que probar: la única Q-sucesión es $\{s\}$, que no termina en G. (Nótese que $s \in P \Rightarrow s \notin G$).

Sea $R(s) \neq \emptyset$:

Por definición de núcleo:	$R(s) \cap K = \emptyset$
por definición de juego:	$R(s) \cap T = \emptyset$
por definición de estrategia:	$Q(s) \in R(s)$

de donde $Q(s) \notin K \cup T$. Además:

$$Q(s) \notin K \Rightarrow R(Q(s)) \cap K \neq \emptyset \quad \text{(por definición de núcleo)}$$

luego en particular $R(Q(s)) \neq \emptyset$, con lo que la Q-sucesión no termina en $Q(s)$.

Por ser $R(Q(s)) \cap K \neq \emptyset$, podemos suponer que el contrincante elige un elemento $s_1 \in R(Q(s)) \cap K$; el proceso puede repetirse con s_1, puesto que $s_1 \in K \cap T$. ■

2.34.- *Teorema.*

Con la notación anterior:

$$\forall s \quad [s \in T - K \Rightarrow \text{existe estrategia } \textit{no-perdedora} \text{ para } s]$$

esto es, es posible evitar que gane el contrincante.

Prueba:

Sea Q cualquier estrategia definida de forma que si $s \notin K$, entonces $Q(s) \in K$ (lo que es posible por definición de núcleo).
$s \in T - K \Rightarrow R(s) \neq \emptyset$; así, será $Q(s) \in K - T$.

Si $Q(s) \in G$, entonces hemos ganado ($Q(s) \notin T \Rightarrow Q(s) \notin P$).

Si $Q(s) \notin G$, pero $R(Q(s)) = \emptyset$, entonces se termina en empate.

Si $R(Q(s)) \neq \emptyset$, entonces $R(Q(s)) \cap K = \emptyset$, luego $R(Q(s)) \subseteq T - K$. Supongamos que el contrincante elige s_1 de manera que $s_1 \in R(Q(s))$; será $s_1 \in T - K$, con lo que podemos repetir el proceso. ■

2.35.- *Corolario.*

Con las notaciones anteriores:

$E - R^{-1}[E] = G \cup P \Rightarrow \forall s \in T - K$ existe estrategia ganadora para s.

Prueba:

Consecuencia inmediata de 2.34. ∎

2.36.- *Teorema.*

Con las notaciones anteriores:

$$E - R^{-1}[E] = G \cup P \Rightarrow T - K = V.$$

Prueba:

Por el corolario anterior, y por 2.23: $T - K \subseteq V$.
Sea $s \in V$; por 2.24: $R(s) \cap (K - T) \neq \emptyset$, de donde $s \in T - K$. ∎

CAPITULO II.- GRAFOS Y/O.

∮.1.- INTRODUCCION.

∮.2.- DEFINICION DE GRAFO Y/O. TEOREMA DE EQUIVALENCIA CON JUEGOS.

∮.3.- ARBOLES Y/O. ARBOLES ALTERNADOS.

§.1.- INTRODUCCION.

Los motores de inferencia de los sistemas expertos son programas que deben extraer conclusiones a partir de una base de conocimientos. Cuando estos conocimientos se expresan mediante cláusulas, y la estrategia usada es de las llamadas "hacia atrás", la representación del proceso deductivo se corresponde con los llamados grafos Y/O (en inglés grafos And/Or). La estrategia "hacia atrás", opera de la siguiente forma: toma la pretendida conclusión y busca en la parte derecha de cada cláusula, para ver si coincide. Para las reglas en las que se produzca la coincidencia anterior, toma su parte izquierda como nueva (ó nuevas) pretendidas conclusiones. El proceso termina en hipótesis de nuestra base ó no. En el primer caso, la conclusión es válida, y en el segundo caso no.

En el Capítulo I hemos estudiado de manera formal los juegos y las estrategias para juegos, también llamadas estrategias alternadas. En este Capítulo vamos a realizar un proceso análogo para los grafos Y/O.

También trataremos en este capítulo la similitud entre ambas representaciones, mediante reglas y mediante grafos. A tal fin, en el capítulo I, formalizamos los conceptos de juego interpretable en grafos, y el de isomorfismo de grafos, que nos permite el paso de una estrategia válida en uno de ellos a una estrategia válida en su imagen isomorfa. Todo este esfuerzo de formalización, lo consideramos necesario para el rigor, en el desarrollo de nuestra idea básica expuesta en la introducción; y que era, expresada en conceptos dados en el capítulo I y en este capítulo II : de un problema de razonamiento automático pasamos al grafo Y/O correspondiente; se considera este grafo como el grafo básico de un juego interpretable en grafos Y/O (o como uno isomorfo).

Por último, en capítulos posteriores hablaremos de la aplicación de la técnica α-β a este grafo de un juego interpretable en grafos Y/O, y del traslado de dicha aplicación al problema original.

Antes de definir con precisión los conceptos mencionados en las lineas precedentes, vamos a presentar un sencillo ejemplo, que

nos irá ilustrando los resultados desarrollados en el presente capítulo.

1.1.- *Ejemplo.*

Supongamos que sabemos que:

C1 : Los que tienen buen sueldo (BS) y buenos amigos (BA), son afortunados (AF).

C2 : Los afortunados, si son cultos (CU), son personas estupendas (PE).

C3 : A las personas estupendas se les puede prestar dinero (PD).

C4 : Las buenas personas (BP) con buen sueldo, merecen confianza (MC).

C5 : A los que tienen buen sueldo y son merecedores de confianza, se les puede prestar dinero.

C6 : Las buenas personas tienen buenos amigos.

Podemos formalizar estas conclusiones mediante las "reglas de conocimiento" (son todas cláusulas):

$$R1 : BS \wedge BA \Rightarrow AF$$

$$R2 : AF \wedge CU \Rightarrow PE$$

$$R3 : PE \Rightarrow PD$$

$$R4 : BP \wedge BS \Rightarrow MC$$

$$R5 : BS \wedge MC \Rightarrow PD$$

$$R6 : BP \Rightarrow BA$$

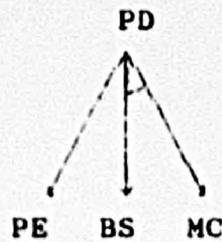
La idea de la estrategia "hacia atrás" es la de división en subproblemas. Por consiguiente, para saber a quién se le puede prestar dinero, se observarán, como ya hemos dicho en párrafos anteriores, las reglas que produzcan PD como conclusión.

De esta observación se deduce que:

PD se tiene si se tiene PE (por la regla R3)
ó si se tiene BS y MC (por la regla R5).

Esta situación puede representarse por

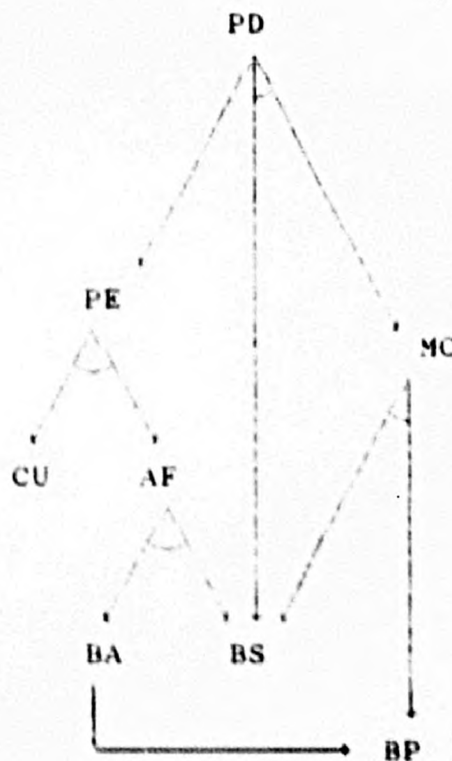
Figura 1.1.1.



donde el lazo entre los arcos significa la conjunción, y la la existencia de varios arcos significa la disyunción.

La representación completa de nuestras reglas, con el "objetivo" PD sería:

Figura 1.1.2.



Pues bien, éste es un ejemplo de lo que denominaremos grafo Y/O.

§.2.- DEFINICION DE GRAFO Y/O. TEOREMA DE EQUIVALENCIA CON JUEGOS.

2.1.- *Definición.*

Un *Grafo* Y/O es un par $\langle N,S\rangle$ donde N es un conjunto finito no vacío, cuyos elementos se llamarán nodos del grafo, y $S \subseteq N \times \mathcal{P}(N)$ ($\mathcal{P}(N)$ = el conjunto de los subconjuntos de N) tales que:

i) La relación que notamos σ, definida en N por

$$n\sigma m \leftrightarrow \exists k \geq 1 \;\; \exists\, n_0,n_1,\ldots,n_k \in N$$

$$n_0 = n \;\wedge\; n_k = m \;\wedge\; [1 \leq i \leq k \rightarrow n_i \in \bigcup S(n_{i-1})]$$

es un orden parcial.

ii) $\dot{\exists} n \in N$ llamado nodo raíz ó inicial, tal que

$$\forall m \in N \quad , \quad n \notin \bigcup S(m).$$

2.2.- *Nota.*

Puede definirse grafo Y/O sin imponer la finitud de N. Aquí la imponemos porque para nuestras aplicaciones (bases de conocimientos) N será siempre finito.

2.3.- *Ejemplo.*

En el ejemplo descrito en 1.1, se tiene:

N = {PD,PE,CU,AF,MC,BA,BS,BP}

S = {<AF, {BS,BA}>, <PE, {AF,CU}>, <PD, {PE}>,
<MC, {BP,BS}>, <PD, {BS,MC}>, <BA, {BP}>}

y el nodo raíz es PD.

2.4.- *Definición.*

Los elementos $\langle n, \{n_1,n_2,\ldots,n_k\}\rangle \in S$ se llamarán k-*arcos* del grafo; los 1-arcos $\langle n, \{m\}\rangle$ serán llamados simplemente *arcos*.

2.5.- *Ejemplo.*

Para el ejemplo descrito en 1.1, no hay más que arcos como por ejemplo <PD, {PE}>, y 2-arcos como por ejemplo <MC, {BP,BS}>.

2.6.- *Notas.*

Damos ahora la justificación de las condiciones de la definición 2.1 de grafo Y/O.

La interpretación lógica de un k-arco $\langle n, (n_1, n_2, \ldots, n_k) \rangle$ es la de una cláusula $n_1 \wedge n_2 \wedge \ldots \wedge n_k \rightarrow n$.

La condición i) nos dice que en el grafo Y/O no hay ciclos; la interpretación es que se nos garantiza la no aparición de autorreferencias: para demostrar una fórmula, no tendremos que demostrar la misma fórmula en el proceso. En juegos, la interpretación es distinta; por ejemplo puede corresponder a situaciones de empate.

En cuanto a la condición ii), establece que nuestro motor actuará con un objetivo cada vez.

2.7.- *Nota.*

Ahora, en el estudio de grafos Y/O, el concepto de "camino" en un grafo pierde su sentido; los caminos vienen sustituidos por grafos Y/O, subgrafos del grafo dado. Con precisión:

2.8.- *Definición.*

Sea $\langle N, S \rangle$ un grafo Y/O. Un *g-camino* de $n \in N$ a $D \subseteq N$ es un grafo Y/O $\langle N', S' \rangle$, definido de forma recursiva por:

i) Si $n \in D$, entonces $N' = \{n\}$, $S' = \emptyset$

ii) Si $n \notin D$:

ii.1) Si existe un k-arco $\langle n, (n_1, \ldots, n_k) \rangle$ tal que para cada i, $1 \leq i \leq k$, exista un g-camino $\langle N'_i, S'_i \rangle$ de n_i a D, entonces

$$N' = \{n\} \cup \left(\bigcup_{i=1}^{k} N'_i \right)$$

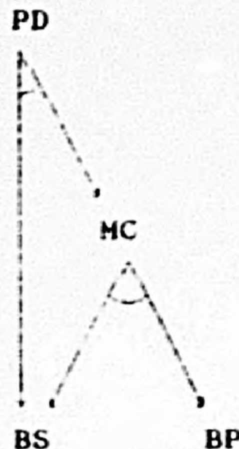
$$S' = \{\langle n, (n_1, \ldots, n_k) \rangle\} \cup \left(\bigcup_{i=1}^{k} S'_i \right)$$

ii.2) Si no, no existe g-camino desde el nodo n al conjunto de nodos D.

2.9.- *Ejemplo.*

En nuestro ejemplo, para ver si a alguien que sea buena persona y tiene buen sueldo se le puede prestar dinero, se podría "razonar":

Figura 2.9.



y como BS y BP son "datos", responderemos afirmativamente.

Traducido a grafos Y/O, esto quiere decir que existe un g-camino de n = PD a D = {BS,BP}. Construimos a continuación dicho g-camino:

> PD ∉ D, y tenemos un 2-arco, <PD, {BS,MC}>. Comprobemos si para los nodos BS y MC existen g-caminos hasta D.
> Para el nodo BS tenemos, BS ∈ D, luego sí existe g-camino de BS a D que es: <{BS},∅>.
> Para el nodo MC tenemos, MC ∉ D, y tenemos un 2-arco, <MC, {BS,BP}>. Comprobemos si para los nodos BS y BP existen g-caminos hasta D.
> Para el nodo BS, sí existe un g-camino hasta D <{BS},∅>.
> Para el nodo BP, también se tiene un g-camino hasta D, <{BP},∅}.

Por tanto, el g-camino del nodo PD al conjunto D = {BS, BP} es el grafo Y/O <N',S'> donde:

N'= {PD,BS,MC,BP}   S'= {<PD, {BS,MC}>, <MC, {BS,BP}>}

Nótese por ejemplo, que, tomando el otro arco <PD, {PE}> no hubiéramos alcanzado la solución, debido a que no existe un g-camino de CU a {BS,BP}.

2.10.- *Teorema.*

Sea $\langle E,R,T,G,P\rangle$ un juego. Consideramos el grafo Y/O $\langle E,S\rangle$ dado por:

$$s \in T \ : \ S(s) = \{\{n\} : n \in R(s)\}$$

$$s \notin T \ : \ S(s) = \{R(s)\}.$$

$\forall s \in T$ se tiene que, existe una estrategia ganadora para s sii existe un g-camino en $\langle E,S\rangle$, de s a G.

Prueba:

$\Longrightarrow$| Condición necesaria.

Consideremos el grafo Y/O $\langle E',S'\rangle$, construido como sigue:

$$E' = \{s,\ s_1 = Q(s)\} \subseteq E$$

$$S' = \{\langle s,\ \{s_1\}\rangle\} \subseteq S.$$

Notemos que $s \in T \Rightarrow s \notin G \Rightarrow Q(s)$ está definida pues si no, $\{s\}$ sería una Q-sucesión que no termina en G, luego Q no sería ganadora para s.

Si $s_1 \in G$, entonces $\langle E',S'\rangle$ sería un g-camino de s a G.

Si $s_1 \notin G$, sea $R(s_1) = \{s_1^1,\dots,s_1^k\}$. Notemos que $R(s_1) \neq \emptyset$ pues si no, tendríamos una Q-sucesión que no termina en G.

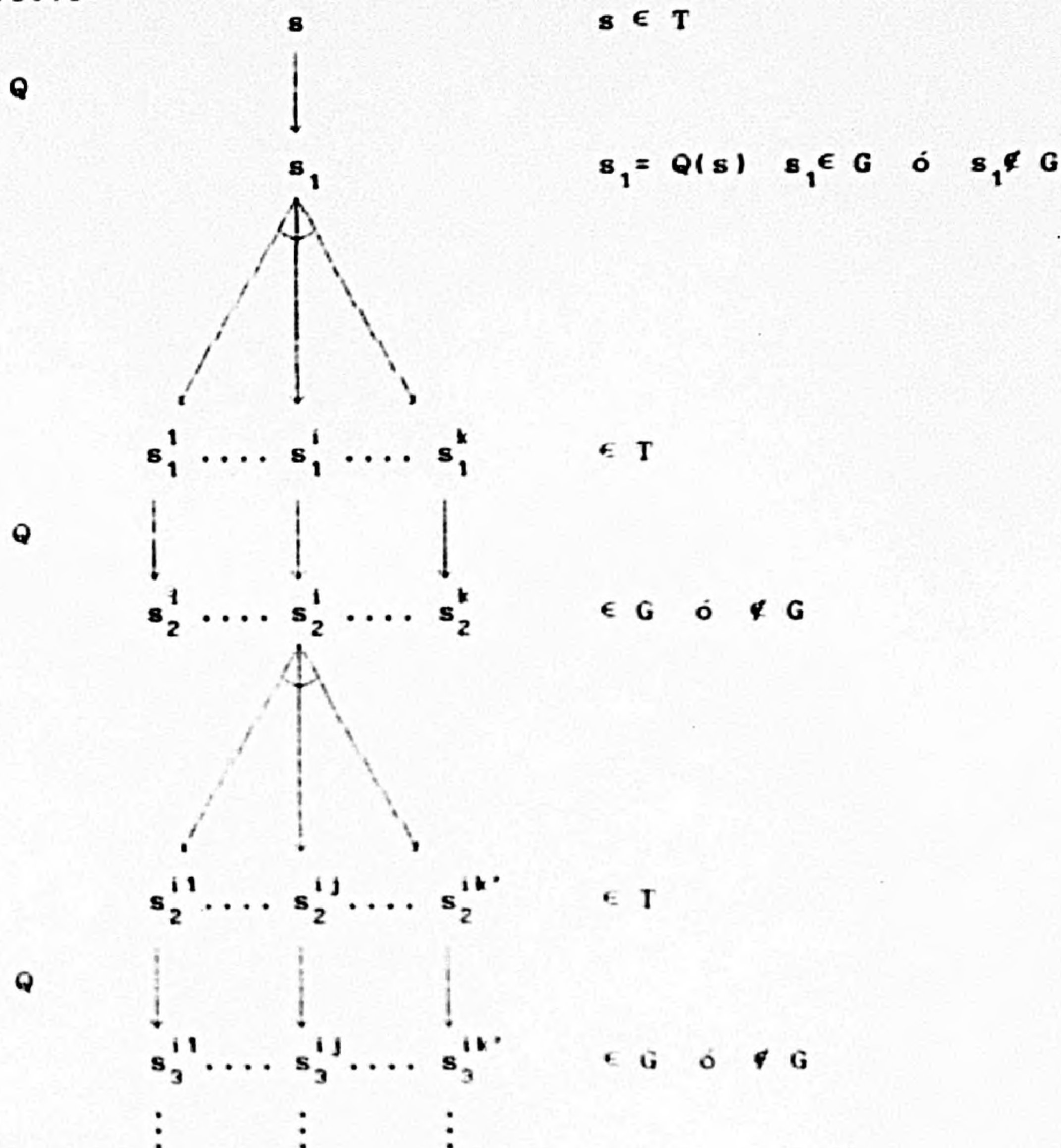
Añadimos a E' los nodos s_1^i, $Q(s_1^i) = s_2^i$, y a S' el k-arco $\langle s_1,\ R(s_1)\rangle$, y los arcos $\langle s_1^i,\ \{s_2^i\}\rangle$, $1 \le i \le k$.

Al igual que con s, Q debe estar definida para los s_1^i, $1 \le i \le k$.

Ahora, con cada s_2^i repetimos el proceso seguido con s_1; si $s_2^i \in G$, no añadimos nada a $\langle E',S'\rangle$ (si todos los $s_2^i \in G$ detenemos el proceso); si no, sea $R(s_2^i) = \{s_2^{i1},\dots,s_2^{ik'}\}$; añadimos los nodos s_2^{ij}, $Q(s_2^{ij}) = s_3^{ij}$, el k'-arco $\langle s_2^i,\ R(s_2^i)\rangle$, y los arcos $\langle s_2^{ij},\ \{s_3^{ij}\}\rangle$, $1 \le j \le k'$.

Es claro que este proceso es finito y termina en nodos de G, como se deduce del hecho de ser toda Q-sucesión finita y terminar en G. Es claro también que el proceso termina con un g-camino de s a G.

Figura 2.10.1



⟹ Condición suficiente.

Sea ahora $\langle E',S' \rangle$ un g-camino en $\langle E,S \rangle$, de s a G. Sea Q una estrategia, definida de cualquier modo, con tal de que

$$Q(s') = s'' \quad \text{si} \quad s' \in T \cap E', \quad \langle s', \{s''\} \rangle \in S'.$$

Veamos que se trata de una estrategia ganadora para s. la idea va a ser, ver que cualquier Q-sucesión para s está "incluida" en $\langle E',S' \rangle$.

Cualquier Q-sucesión para s es de la forma:

$$(s = \underbrace{s_0, s_1}_{Q}, \underbrace{s_1^{i}, s_2^{i}}_{Q}, \underbrace{s_2^{ij}, s_3^{ij}}_{Q}, \underbrace{s_3^{ijk}, \ldots}_{Q})$$

El comportamiento de estos nodos se resume según los dos siguientes casos, según se observa en la representación gráfica de la Q-sucesión anterior, dada en la figura 2.10.2.

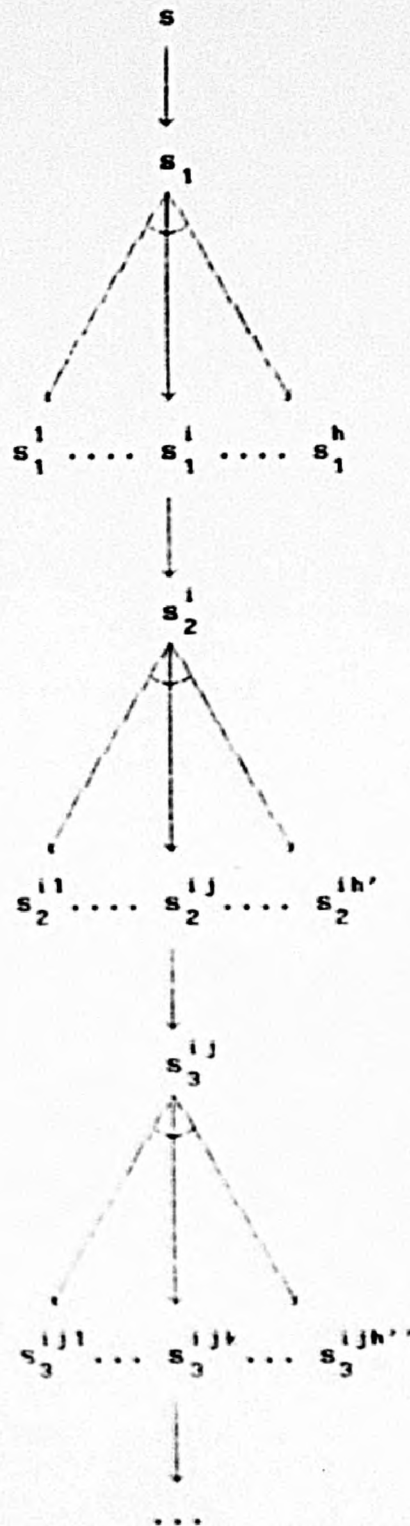
$s_n^{i_1 \ldots i_n}$: Estos nodos son de T, luego no están en G; deberán tener sucesores por la definición de g-camino; su sucesor es justamente

$$Q(s_n^{i_1 \ldots i_n}) = s_{n+1}^{i_1 \ldots i_n}.$$

$s_{n+1}^{i_1 \ldots i_n}$: Estos nodos no son de T; si son de G, entonces la Q-sucesión es finita y termina en G; si no, deberá tener sucesores (debido a la definición de g-camino), luego la sucesión no puede terminar en él.

Así, si la Q-sucesión termina, lo hace en G. Pero es que tiene que terminar, pues el conjunto de nodos es finito (y el grafo es acíclico), luego no admite caminos infinitos. ■

Figura 2.10.2.



§.3.- ARBOLES Y/O. ARBOLES ALTERNADOS.

3.1.- *Definición.*

Sea $\langle N,S\rangle$ un grafo. Se dice que $\langle N,S\rangle$ es un árbol si

$$\forall m \in N \;\; [m \neq \text{nodo raíz} \Rightarrow \exists! n \in N \; [nSm]]$$

Análogamente, si $\langle N,S\rangle$ es un grafo Y/O, se dice que $\langle N,S\rangle$ es un árbol Y/O si

$$\forall m \in N \;\; [m \neq \text{nodo raíz} \Rightarrow \exists! n \in N \; [m \in \bigcup S(n)]]$$

Tal nodo n se llama antecesor inmediato o *padre* del nodo m.

3.2.- *Definición.*

Sea $\langle N,S\rangle$ un árbol ó un árbol Y/O. La profundidad es una función $Prof : N \longrightarrow \mathbb{N}$, con $\mathbb{N}$ el conjunto de los números naturales, definida recursivamente por

$$Prof(n) = \begin{cases} 0 \;\; \text{si} \;\; n = \text{nodo raíz de } \langle N,S\rangle \\ 1 + Prof(\text{padre de } n) \;\;\; \text{en otro caso} \end{cases}$$

3.3.- *Definición.*

Decimos que el par $\langle N,S\rangle$ es un *Arbol de juego ó Arbol Alternado*, si $\langle N,S\rangle$ es un árbol Y/O y además:

$$\forall n \in N \;\; [Prof(n) \text{ es par} \Rightarrow \forall \langle n,A\rangle \in S \; [A \text{ es unitario}]], \text{ y}$$

$$\forall n \in N \;\; [Prof(n) \text{ es impar} \Rightarrow \exists! A \in \mathcal{P}(N) \; [\langle n,A\rangle \in S]].$$

3.4.- *Nota.*

Informalmente, esto quiere decir que

$\forall n \in N$ si $Prof(n)$ es par, entonces n está unido con sus inmediatos sucesores exclusivamente por 1-arcos, y

si $Prof(n)$ es impar, entonces n está unido a sus inmediatos sucesores exclusivamente por un k-arco.

3.5.- *Notas.*

Nótese que lo que hemos definido como árbol de juego, se adecúa al desarrollo normal de una jugada. El jugador que tiene

el turno analizará sus posibles movimientos como distintas alternativas que le llevan a distintos estados en los que el turno lo tiene el adversario. Para elegir la mejor de sus alternativas, para cada una de ellas, deberá tener en cuenta todas las posibles réplicas del adversario. Esto se repite hasta un determinado nivel de profundidad, ó hasta que se llege a estados ganadores ó perdedores (estados sin sucesores).

Así, el estado del jugador que tiene el turno, lo uniremos con sus respectivos sucesores mediante 1-arcos, que representan las alternativas que tiene. También de acuerdo a lo dicho antes, cada sucesor de este estado inicial lo uniremos con sus respectivos sucesores mediante un k-arco que expresa la conjunción de todas las posibles réplicas. El proceso se repite hasta una profundidad determinada ó hasta estados sin sucesores.

3.6.- *Definición.*

Dado un grafo Y/O, <E,S>, llamaremos *Arbol Y/O asociado* al grafo dado, al árbol

<E',S'> = GRA-ARB (<E,S>)

donde GRA-ARB es el algoritmo siguiente:

```
GRA-ARB (<E,S>)
        E' := {<nodo inicial,1>}
        S' := Ø
        NA := E'          {NA seran los "abiertos"}
        CICLO.
CICLO ()
        SI  NA = Ø  ENTONCES FIN:  <E',S'>
        <nod,num> := un elemento cualquiera de NA
        sucs := sucesores de nod (mediante S)
        NA := NA - {<nod,num>}
        SI  sucs = Ø  ENTONCES  CICLO
        Para cada A ∈ sucs:
             N* := Ø
             Para cada m ∈ A:
                  N* := N* U {<m,1+índice de m en E'>}
             E' := E' U N*
             NA := NA U N*
             S' := S' U {<<nod,num>, N*>}
        CICLO.
```

El índice de m en E' es:

0, si $\forall \langle n,i \rangle \in E'$ es $n \neq m$

i, si $\langle m,i \rangle \in E' \ \wedge \ \langle m,i+1 \rangle \notin E'$.

El programa LISP correspondiente es el siguiente:

```
(de GRA-ARB (e s)
    (setq              ;el nodo inicial debe ir el primero de E
        ee (list (list (car e) 1))    ;ee ss sera el resultado
        ss ()
        na (list (list (car e) 1)))  ;na son los "abiertos"
    (CICLO) )
;
(de CICLO ()
    (if (null na)
        (list  ee ss)
        (setq no-nu (car na) sucs (SUCES (caar na) s) na (cdr na))
        (if (null sucs)
            (CICLO)
            (mapcar 'AUX-1 sucs) (CICLO)))))
;
(de AUX-1 (a)
    (setq n* ())
    (mapcar 'AUX-2 a)
    (setq ee (append ee n*)
         na (append na n*)
         ss (append ss (list (list no-nu n*))))))
;
(de AUX-2 (m)
    (setq n* (append n* (list (list m (+ 1 (INDICE m ee 0))))))))
;
(de INDICE (m lista vp)
            ;la funcion INDICE nos da 0 si m no está en lista,
            ;y si está, da el máximo índice
    (if (null lista)  vp
        (let ((nodo (caar lista))  (numero (cadar lista)) )
            (if (and (equal m nodo) (< vp numero))
                (INDICE m (cdr lista) numero)
                (INDICE m (cdr lista) vp)))))
```

```
(de SUCES (nod lista)
    (cond ((null lista) ())
          ((equal nod (caar lista))
                 (cons (cadar lista) (SUCES nod (cdr lista))))
          (t (SUCES nod (cdr lista)))))
```

Más adelante se verá el efecto de este algoritmo sobre un grafo.

3.7.- *Definición.*

Dado un grafo Y/O, <E,S>, llamaremos *Arbol Alternado asociado* al grafo dado, al árbol

$$\langle E'',S''\rangle = \text{AR-AA}\ (\text{GRA-ARB}\ (\langle E,S\rangle))$$

donde AR-AA es el algoritmo siguiente:

```
AR-AA (<E',S'>)          [E' y S' vienen dados por GRA-ARB]
        E'' := Ø
        S'' := Ø
        NA := {<0,nodo inicial de E'>}
        CICLO2.
```

```
CICLO2 ( )
        SI  NA = Ø  ENTONCES FIN:  <E'',S''>
        pn := un elemento de NA, de los de menor profundidad
                                                        [ver 3.8.1]
        E'' := E'' U {no-nu}
        sucs2 := sucesores de no-nu                     [ver 3.8.2]
        NA := NA - {pn}
        SI  sucs2 = Ø  ENTONCES  CICLO2
        SI  prof es par  ENTONCES
             Para cada A ∈ sucs2:
                  SI  A es unitario, A = {m}, ENTONCES
                       S'' := S'' U {<no-nu, A>}
                       NA := NA U {<1+prof,m>}
                  SI  A no es unitario, ENTONCES
                       ind := índice de nod en E'
                       E' := E' U {<nod,1+ind>}        [ver 3.8.3]
                       S'' := S'' U {<no-nu, {<nod,1+ind>}>}
                                  U {<<nod,1+ind>, A>}
                       E'' := E'' U {<nod,1+ind>}
                       Para cada m ∈ A:
                            NA := NA U {<prof+2,m>}
        SI  prof es impar, ENTONCES
             S'' := S'' U {<no-nu, {car (sucs2)}>}
             SI  sucs2 es unitario, sucs2 = {A}, ENTONCES
                  Para cada m ∈ A:
                       NA := NA U {<prof+1,m>}
             SI  sucs2 no es unitario, ENTONCES
                  Para cada A ∈ cdr (sucs2):
                       ind := índice de nod en E'
                       E' := E' U {<nod,1+ind>}
                       E'' := E'' U {<nod,1+ind>}
                       S'' := S'' U {<PADRE(no-nu),{<nod,1+ind>}>}
                                  U {<<nod,1+ind>,A>}
                  Para cada A ∈ sucs2:
                       Para cada m ∈ A:
                            NA := NA U {<1+prof,m>}
        CICLO2.
```

PADRE (no-nu)   [se entiende que en S']
    (no-nu') tal que no-nu $\in$ $\bigcup$ S'(no-nu')

El correspondiente programa LISP es:

```
(de AR-AA (ee ss)
    (setq  eee ()  sss ()  na (list (list 0 (car ee))))
    (CICLO2))                    ;pn = (prof no-nu) = (prof (nod num))
;
(de CICLO2 ()
  (if (null na)
      (list eee sss)
      (setq pn (car na) prof (caar na) no-nu (cadar na)
            nod (caadar na))
      (setq eee (append eee (cdr pn))
            sucs2 (SUCES2 no-nu ss)  na (cdr na) )
      (if (null sucs2)
          (CICLO2)
          (if (evenp prof)
              (mapcar 'AUX-11 sucs2)
              (setq sss
                 (append sss (list (list no-nu (car sucs2)))))
              (if (= 1 (length sucs2))
                  (setq na
                     (append na (mapcar 'PROF-1 (car sucs2))))
                  (mapcar 'AUX-22 (cdr sucs2))
                  (setq na
                   (append na
                     (apply 'append (mapcar
                                    '(lambda (a)(mapcar 'PROF-1 a))
                                     sucs2)) ) ) ) )
              (CICLO2) ) ) )
;
(de SUCES2 (no-nu lista)
    (cond  ((null lista) ())
           ((equal no-nu (caar lista))
            (cons (cadar lista) (SUCES2  no-nu  (cdr  lista))))
           (t (SUCES2 no-nu (cdr lista))) ) )
```

```
(de AUX-11 (a)
  (if (= (length a) 1)
      (setq sss (append sss (list (list no-nu a)))
            na  (append na (list (cons (+ 1 prof) a))) )
      (setq ind (INDICE nod ee 0)
            ee (append ee (list (list nod (1+ ind))))
            sss (append sss
                  (list (list no-nu (list (list nod (1+ ind))))
                        (list (list nod (1+ ind)) a)))
            eee (append eee (list (list nod (1+ ind))))
            na (append na (mapcar 'PROF-2 a)) ) ) )
;
(de PROF-2 (m)
   (list (+ 2 prof) m))
;
(de AUX-22 (a)
     (setq ind (INDICE nod ee 0)
           ee (append ee (list (list nod (1+ ind))))
           eee (append eee (list (list nod (1+ ind))))
           sss (append sss
                     (list (list (PADRE no-nu sss)
                                 (list (list nod (1+ ind))))
                           (list (list nod (1+ ind)) a))))))
;
(de PROF-1 (m)
   (list (+ 1 prof) m))
;
(de PADRE (n lista)
  (any '(lambda (x) (if (member n (cadr x)) (car x) ())) lista))
```

3.8.- *Notas.*

3.8.1.- NA denota el conjunto de nodos que no se han expandido. Tal como crece NA, puede tomarse el primer elemento. La representación será pn = <prof,no-nu> = <prof,<nod,num>>. Nótese que como figura en la definición 3.6, GRA-ARB debe de empezar al tratar con el nodo inicial del grafo. AR-AA también debe de empezar a tratar el nodo inicial del grafo. En el caso de GRA-ARB, hemos de proporcionar el nodo inicial como primer nodo del

conjunto E. En el caso de AR-AA, la salida de GRA-ARB, garantiza que el nodo inicial es el primer elemento de E'.

3.8.2.- Naturalmente, se trata de sucesores en E', no en E.

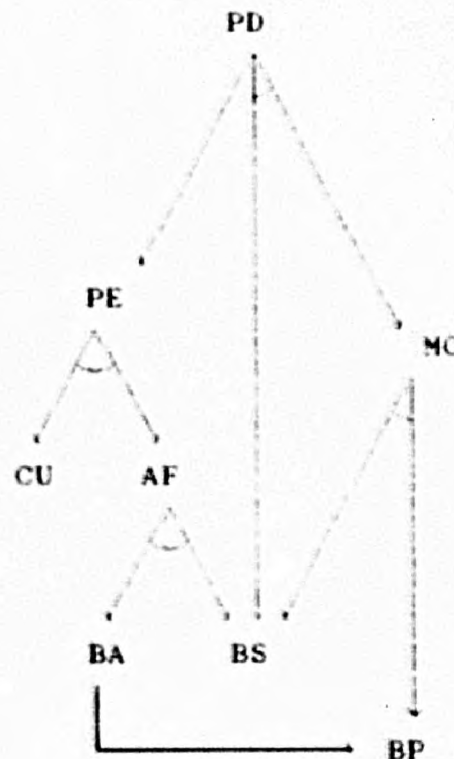
3.8.3.- E' se modifica únicamente a efectos de actualización del índice.

3.8.4.- La función INDICE, solo aparece codificada en el programa GRA-ARB, por ser coincidente en ambos programas GRA-AR y AR-AA.

3.9.- *Notas.*

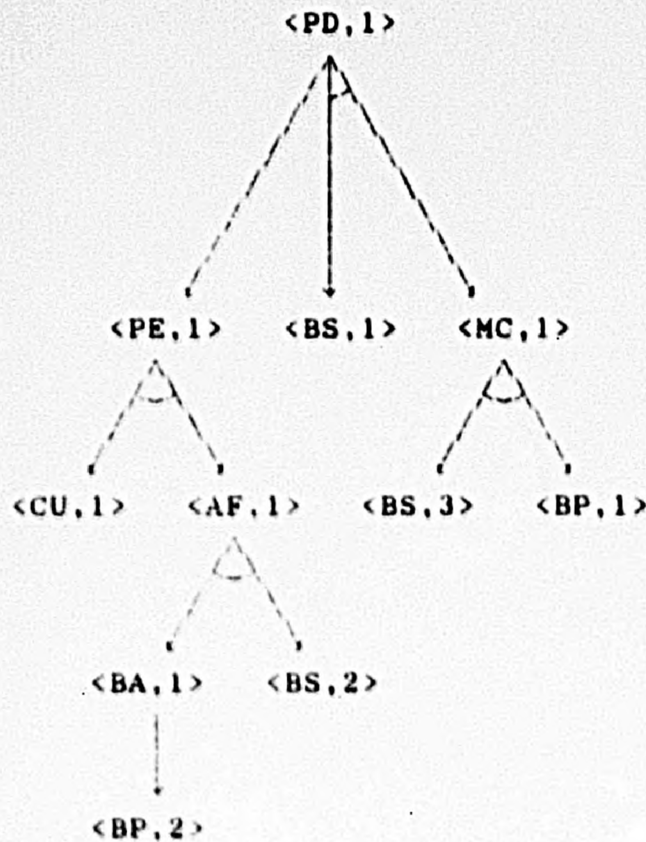
La idea de estos algoritmos es muy simple. GRA-ARB hace "copias" de los nodos con más de un antecesor, para conseguir que cada nodo, excepto el raíz, tenga un solo antecesor. Por ejemplo con el grafo

Figura 3.9.1.



el algoritmo GRA-ARB nos devuelve:

Figura 3.9.2.



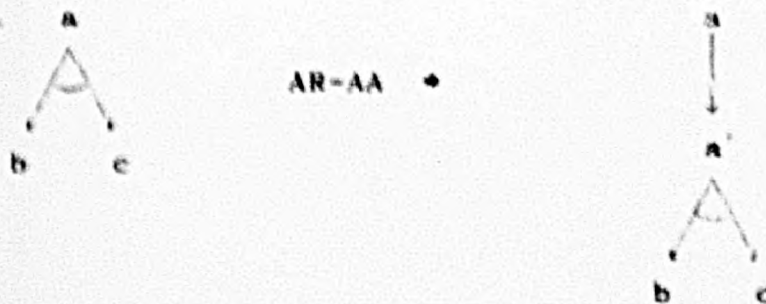
Por su parte, el algoritmo AR-AA consigue un árbol alternado a partir de éste, sin más que añadir arcos suplementarios donde sea preciso, esto es:

1) para cada $\langle n, A\rangle \in S'$, si la profundidad de n es par y A no es unitario, se sustituye el k-arco $\langle n, A\rangle$ por el 1-arco $\langle n, \{n'\}\rangle$, junto con el k-arco $\langle n', A\rangle$, donde n' es una copia de n, y

2) para cada n de profundidad impar, si es origen de más de un k-arco, por ejemplo, $\langle n, A_1\rangle, \ldots, \langle n, A_q\rangle$, se hacen q-1 copias de n para que cada una de ellas sea origen de un único k-arco.

Ejemplo de la situación 1: (a de profundidad par)

Figura 3.9.3.

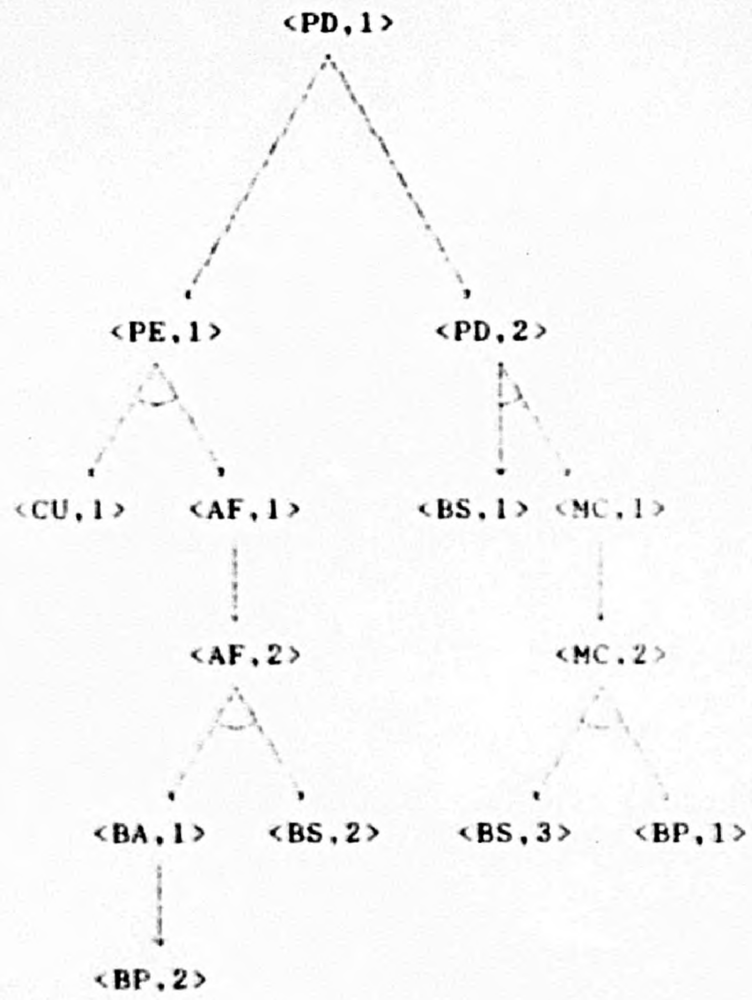


y de la situación 2: (a de profundidad impar)

Figura 3.9.4.

Naturalmente, estas sustituciones deben hacerse comenzando por los nodos de menor profundidad, porque cada sustitución altera la profundidad de los sucesores. En el ejemplo que veníamos considerando, el árbol alternado resultante será:

Figura 3.9.5.



En lo que sigue, vamos a conservar esta notación:

<E,S> para un grafo Y/O cualquiera,

<E',S'> para el árbol Y/O asociado, y

<E'',S''> para el árbol alternado asociado correspondiente.

Considerando sólo los nodos, cada n ∈ E da lugar, en el paso de E a E′, a un conjunto de nodos de E′: <n,1>,...,<n,k>, k ≥ 1; a su vez, cada uno de estos nodos de E′, en el paso de E′ a E′′, puede eventualmente dar lugar a que se incremente el número k de copias de n. Si D ⊆ E, llamaremos D′ al conjunto de los nodos de E′ que provienen de nodos de D en el paso de E a E′, y D′′ al conjunto de los nodos de E′′ que provienen de nodos de D′ en el paso de E′ a E′′.

3.10.- *Teorema.*

Con las notaciones anteriores, y siendo NI el nodo inicial de <E,S>, las afirmaciones siguientes son equivalentes:

(1) ∃ g-camino de NI a D en <E,S>

(2) ∃ g-camino de <NI,1> a D′ en <E′,S′>

(3) ∃ g-camino de <NI,1> a D′′ en <E′′,S′′>.

Prueba:

Antes de la demostración propiamente dicha, veamos la situación con el ejemplo que venimos considerando, tomando D = {BS,BP}. (Los g-caminos aparecen marcados con *)

Figura 3.10.1.

D = {BS,BP}

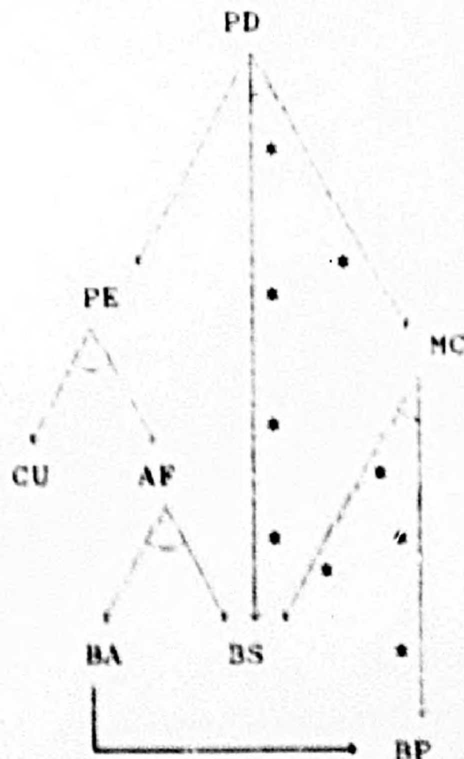


Figura 3.10.2.

D' = {<BS,1>,<BS,2>,
<BS,3>,<BP,1>,
<BP,2>}

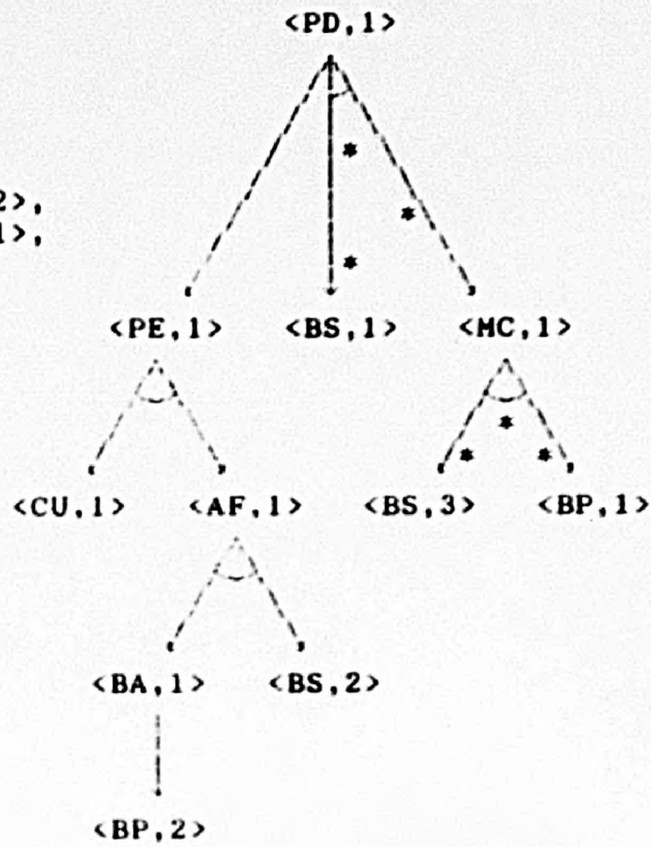
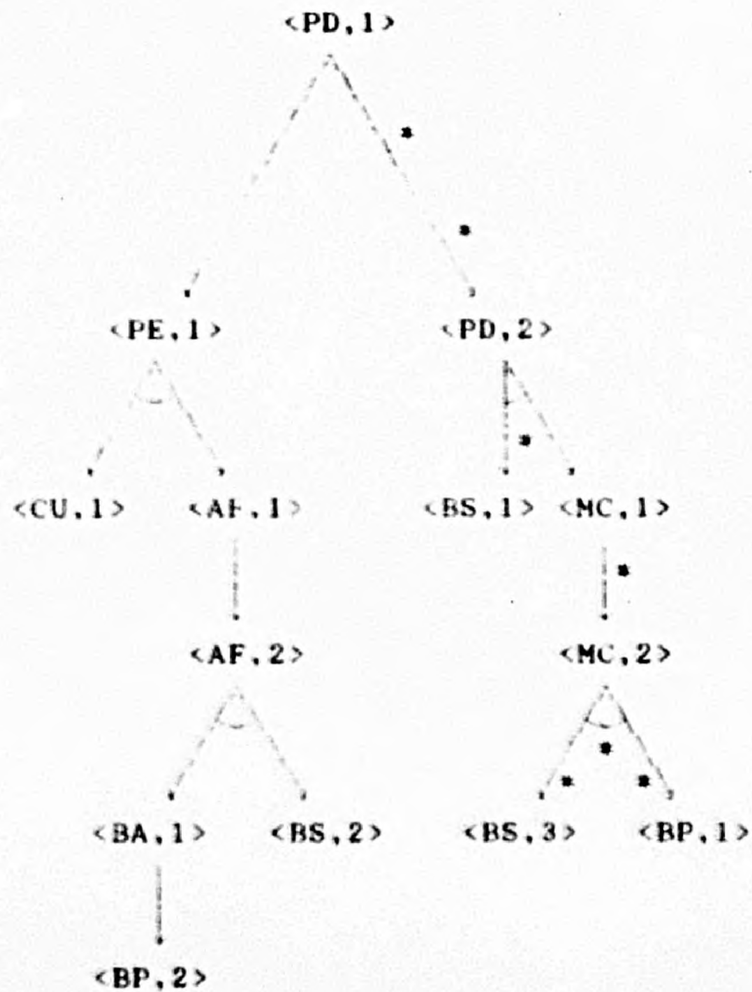


Figura 3.10.3.

D'' = D'



(1) ⇒ (2)| Sea <M,T> un g-camino de NI a D en <E,S>. El algoritmo:

$M' := \emptyset$

$T' := \emptyset$

Para cada $\langle n, \{n_1,\dots,n_k\}\rangle \in T$, se consideran los k-arcos de S' correspondientes a los mismos nodos, independientemente de la numeración. Si estos arcos son:

$$\langle\langle n,i_1\rangle, \{\langle n_1,i_{11}\rangle,\dots,\langle n_k,i_{k1}\rangle\}\rangle$$
$$\dots\dots\dots$$
$$\langle\langle n,i_p\rangle, \{\langle n_1,i_{1p}\rangle,\dots,\langle n_k,i_{kp}\rangle\}\rangle$$

entonces hacemos:

$$M' := M' \cup \{\langle n,i_1\rangle,\langle n_1,i_{11}\rangle,\dots,\langle n_k,i_{k1}\rangle,\dots$$
$$\dots, \langle n,i_p\rangle,\langle n_1,i_{1p}\rangle,\dots,\langle n_k,i_{kp}\rangle\}$$

$$T' := T' \cup \{\langle\langle n,i_1\rangle, \{\langle n_1,i_{11}\rangle,\dots,\langle n_k,i_{k1}\rangle\}\rangle,$$
$$\dots\dots\dots$$
$$\langle\langle n,i_p\rangle, \{\langle n_1,i_{1p}\rangle,\dots, \langle n_k,i_{kp}\rangle\}\rangle\}$$

calcula <M',T'>, que es claro que es un g-camino de <NI,1> a D' en <E',S'>.

(2) ⇒ (1)| Sea <M',T'> un g-camino de <NI,1> a D' en <E',S'>. El algoritmo:

$M := \emptyset$

$T := \emptyset$

Para cada $\langle\langle n,i\rangle, \{\langle n_1,i_1\rangle,\dots,\langle n_k,i_k\rangle\}\rangle \in T'$

$M := M \cup \{n,n_1,\dots,n_k\}$

$T := T \cup \{\langle n, \{n_1,\dots,n_k\}\rangle\}$

calcula <M,T>, que es claro que es un g-camino de NI a D en <E,S>. Nótese que estas uniones son conjuntistas (no como *append* en LISP), es decir, que si por ejemplo el k-arco $\langle n, \{n_1,\dots,n_k\}\rangle$ ya estuviese en T, T no se altera en el paso correspondiente del algoritmo.

(2) ⇒ (3) Sea <M',T'> un g-camino de <NI,1> a D' en <E',S'>. Es claro que, salvo diferencias en los índices (pues el conjunto es distinto), AR-AA (<M',T'>) es un g-camino de <NI,1> a D'' en <E'',S''>.

(3) ⇒ (2) Sea <M'',T''> un g-camino de <NI,1> a D'' en <E'',S''>. El algoritmo:

M' := M''
T' := T''
Para cada situación <<n,i>, {<n,i'>}>, <<n,i'>, A> ∈ T''
hacemos
M' := M' - {<n,i'>}
T' := T' ∪ {<<n,i>, A>}
- {<<n,i>, {<n,i'>}>, <<n,i'>, A>}

calcula <M',T'>, que es, salvo diferencias en los índices, un g-camino de <NI,1> a D' en <E',S'>. Ahora, el orden en que se hagan las sustituciones anteriores, es irrelevante. ■

CAPITULO III.- PROCEDIMIENTOS MIN-MAX Y ALFA-BETA.

§.1.- PROCEDIMIENTO MIN-MAX.

§.2.- PROCEDIMIENTO ALFA-BETA.

§.1.- PROCEDIMIENTO MIN-MAX.

1.1.- Descripción del procedimiento.

1.1.1.- *Notas.*

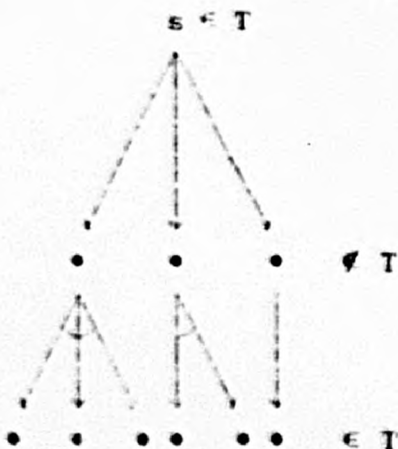
Nos vamos a centrar en juegos <E,R,T,G,P> interpretables en grafos, con grafo de juego <N,R'> y subconjunto $F \subseteq N$. Vamos a buscar estrategias para el jugador que comienza la partida. A este jugador lo denotaremos por jugador MAX, y en general, los estados correspondientes a este jugador (que comienza la partida) vienen dados por el conjunto $T = \{\langle n,1\rangle : n \in N\}$ y los llamamos nodos ó estados MAX. Los estados del adversario los llamaremos nodos ó estados MIN.

Dado <E,R,T,G,P> un juego interpretable en grafos. Consideremos el grafo Y/O, <E,S> construido según el teorema II-2.10, y el árbol alternado asociado (ver definición III-3.7), <E'',S''> = AR-AA (GRA-ARB (<E,S>)).

Si suponemos que el nodo inicial pertenece a T, entonces, por la construcción de <E,S>, AR-AA no es preciso pues ya GRA-ARB (<E,S>) es alternado. En general, esto será perfectamente asumible debido al hecho de que buscamos estrategias para el jugador que comienza el juego, es decir, para estados de T.

El procedimiento Min-Max, pretende la elección de la "mejor" jugada. Este procedimiento trabaja con subgrafos de <E'',S''> construidos a partir de un nodo inicial $s \in T$, lo que notaremos por J(s)

Figura 1.1.1



desarrollados sólo hasta una cierta cota de profundidad: lo que a su vez notaremos J(s,cota). La interpretación de J(s,cota) es la del grafo de la jugada para un nodo s ∈ T, hasta una profundidad igual a cota. Nótese que, de acuerdo con II-3.3 y 3.7, <E'',S''> es un árbol que se denominaba árbol del juego ó árbol alternado del juego <E,R,T,G,P>. J(s,cota) es un subarbol de <E'',S''> cuya interpretación es la del *árbol de la jugada* para un nodo s ∈ T, hasta una profundidad igual a cota. En ocasiones a J(s,cota) simplemente lo denominaremos árbol de juego.

Antes de describir el procedimiento Min-Max, vamos a definir un nuevo concepto que es utilizado tanto por éste como por el procedimiento Alfa-Beta. Este concepto es el de función de evaluación estática.

Nótese también, que como la profundidad del nodo inicial es 0, los nodos MAX tienen profundidad par mientras que los nodos MIN tienen profundidad impar, en el árbol de juego correspondiente. Esto quiere decir, que podemos identificar el jugador a que corresponde un determinado nodo ó estado del juego por la profundidad del nodo.

1.1.2.- *Definición.*

Sea <E,R,T,G,P> un juego interpretable en grafos. Una *función de evaluación estática* es cualquier aplicación $h: N \longrightarrow \mathbb{Z}$ con $\mathbb{Z}$ el conjunto de los números enteros.

1.1.3.- *Nota.*

Una función de evaluación estática nos debe dar un valor de la estimación del nodo. Normalmente este valor será numérico, que para nosotros será suficiente considerar $\mathbb{Z}$. En realidad cualquier conjunto ordenado serviría para los propósitos de evaluación y comparación de nodos. La idea que se persigue es simple:

i) $\forall n \in N$ $h(n) \geq 0$ si n es una posición "favorable" a MAX y tanto mayor cuanto más favorable.

ii) $\forall n \in N$ $h(n) \leq 0$ si n es una posición "desfavorable" a MAX ó lo que es lo mismo favorable a MIN, y tanto menor cuanto más desfavorable.

iii) Normalmente, suele escribirse $h(n) = +\infty$ si $\langle n,0\rangle \in G$
$h(n) = -\infty$ si $\langle n,1\rangle \in P$,

abusando un poco del lenguaje (pues $\pm\infty$ no son elementos de $\mathbb{Z}$).

1.1.4.- *Notas.*

El valor 1 para el indicador de jugador representa al que hemos denotado como jugador MAX. El valor 0 denota al jugador MIN. Como ya hemos observado al final de la nota 1.1.1, este indicador de jugador puede sustituirse por la profundidad del nodo.

Las funciones de evaluación estática, son por lo general, funciones heurísticas, construidas en base a conjeturas que nos hacemos sobre el juego.

1.1.5.- *Procedimiento Min-Max.*

El procedimiento Min-Max, es un procedimiento que permite, a partir de las evaluaciones estáticas de los nodos de último nivel considerado y de los nodos sin sucesores, evaluar los nodos de niveles anteriores hasta llegar al nodo inicial. A partir de esta evaluación propagada, hacemos la elección de la jugada que creemos más ventajosa.

Para elegir qué jugada realiza MAX, desarrollamos el árbol de juego hasta un determinado nivel de profundidad (esto es, establecemos una cota al desarrollo); evaluamos los nodos de dicho nivel mediante la función de evaluación estática y propagamos la evaluación a nodos antecesores hasta llegar al nodo inicial. Esta propagación de la evaluación estática se realiza de la siguiente forma:

Para cada nodo n del árbol de juego J(s,cota), excepto s, se define su valor de evaluación EV por

$$EV(n) = \begin{cases} h(n) & \text{si } S''(n) = \emptyset \text{ ó si profundidad de } n \text{ en } J(s,cota) \text{ es igual a cota} \\ \text{máx } \{ \text{mín } \{ EV(m) : m \in A \} : \langle n, A\rangle \in S'' \} & \text{en otro caso.} \end{cases}$$

La jugada que se escoge es entonces la correspondiente al sucesor de s de mayor valor de evaluación.

Este procedimiento define la siguiente estrategia:

$Q: T'' \longrightarrow E''$, tal que a $s \in T''$ le asocia $Q(s) = s'$ que verifica $s' \in S''(s) \wedge \forall s'' \in S''(s)\ EV(s') \geq EV(s'')$.

1.2.- Algoritmo.

A continuación damos el algoritmo que implementa el procedimiento Min-Max. Partimos de un juego interpretable en grafos <E,R,T,G,P>. Supondremos que estamos en un determinado nodo, que se considera como nodo inicial y que denotaremos por NI, que estamos limitados por una determinada cota de profundidad en la generación del árbol de la jugada para NI, y que contamos con una función de evaluación estática que denotaremos por h. Pretendemos obtener, como paso previo, el árbol de juego correspondiente, para después evaluarlo y así tener la elección de el nodo sucesor al nodo inicial dado que nos sea más favorable. Nótese que esta medida de lo favorable o desfavorable que nos resulta un determinado nodo, va a depender de la función de evaluación estática que se tenga, es decir, la eficiencia del procedimiento Min-Max depende de dicha función. Se plantea así una dificultad adicional proporcionada por el hecho de tener que encontrar buenas funciones de evaluación estática.

En el algoritmo distingimos tres etapas fundamentales:

- Etapa 1.- Generación de nodos, a partir de un nodo inicial.
- Etapa 2.- Evaluación de nodos.
- Etapa 3.- Elección de siguiente nodo, de entre los nodos sucesores del nodo inicial.

A primera vista estas tres etapas son secuenciales, es decir, primero generamos el árbol, después lo evaluamos, y por último, elegimos el siguiente nodo que será el nodo más favorable de entre los sucesores del nodo inicial a partir del que hemos generado el árbol. Sin embargo, de cara a la ejecución del algoritmo, las dos primeras etapas se realizan conjuntamente debido a que la evaluación de nodos va a ir llamando a la

generación de nodos. Es decir, no vamos a generar el árbol completo y despuéь lo vamos a evaluar, sino que generaremos los sucesores del nodo inicial, evaluaremos estos sucesores para lo cual necesitaremos generar nuevos sucesores, y así sucesivamente hasta llegar a la cota de profundidad ó hasta llegar a nodos que no tengan sucesores. Por otra parte la segunda etapa de evaluación de nodos va a ser recursiva, siendo precisamente las condiciones mencionadas en las líneas precedentes de alcanzar la cota de profundidad en la generación del árbol ó de alcanzar nodos sin sucesores las condiciones de parada de la recursividad. Pasamos a continuación a la descripción del algoritmo. Esta recursividad la observaremos mejor en la sintaxis de los programas, ya que la descripción que vamos a dar del algoritmo será secuencial, plasmando cada una de las etapas.

En primer lugar, vamos a dar una descripción más general, que refleja cada una de estas tres etapas, para luego dar una descripción más detallada de cada paso del algoritmo. El orden empleado para describir las tres etapas va a ser inverso al orden en que las hemos enunciado antes. Esto es debido, a que por razones de comprensión de la escritura, preferimos el siguiente orden: en primer lugar nosotros queremos realizar un movimiento de un nodo dado a algún sucesor, para ello necesitamos saber cúal es el sucesor más favorable (generación y evaluación).

Etapa 3.-

Datos de entrada al algoritmo:

NI = nodo inicial

cota = cota de profundidad

H = evaluación estática.

Esta etapa devuelve el siguiente movimiento a realizar.

Tomar NI y asignar profundidad 0.

Calcular sucesores de NI.

Evaluar sucesores de NI.

Ordenar los sucesores de NI según su evaluación.

Elegir como siguiente movimiento el que nos lleva al primer nodo del conjunto de nodos sucesores de NI, ordenado.

Etapa 2.-

Esta etapa realiza las tareas del procedimiento Min-Max. Va a recoger la propagación de la evaluación en el árbol. También determina la generación completa del árbol. Esta etapa devuelve la evaluación de un nodo (i.e. NI).

Tomar un nodo y su profundidad.

Tomar sucesores del nodo.

Si profundidad = cota ó el nodo no tiene sucesores, entonces, calcular el valor de la evaluación estática del nodo en cuestión.

En caso contrario,

Si profundidad = par, entonces, tomar como evaluación del nodo, el valor máximo de las evaluaciones de sus sucesores.

En caso contrario,

Se tiene que profundidad = impar, entonces, tomar como evaluación del nodo, el valor mínimo de las evaluaciones de sus sucesores.

Etapa 1.-

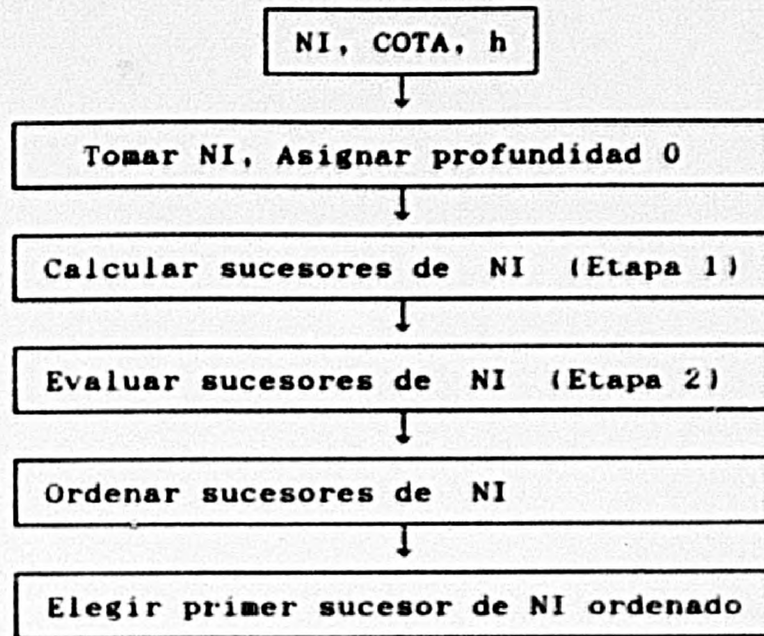
Esta etapa, así como la función de evaluación estática, es particular de cada juego. Debido a que desarrollamos hasta una determinada cota de profundidad, y también al hecho de que hemos de diferenciar a los dos jugadores en la etapa 2, hemos de ir arrastrando la profundidad de cada nodo. Esta etapa devuelve los sucesores de un nodo.

Tomar nodo

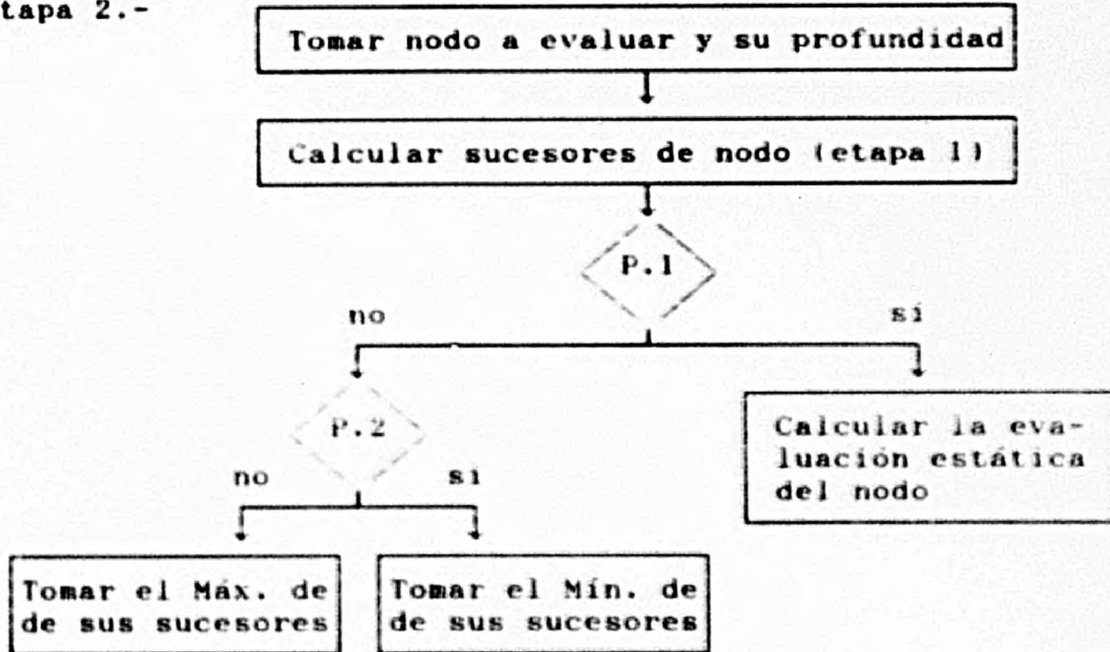
Calcular sucesores del nodo y la profundidad de dichos sucesores.

Esquemáticamente, estas tres etapas las podemos representar por los siguientes organigramas:

Etapa 3.-



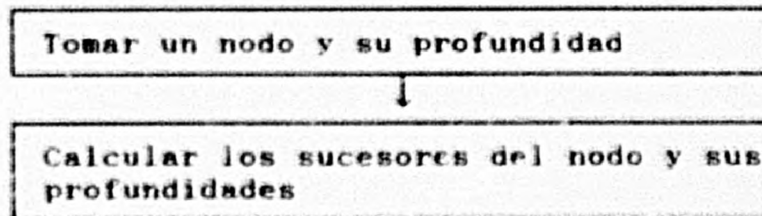
Etapa 2.-



P.1 = Profundidad = cota ó nodo no tiene sucesores

P.2 = profundidad = impar

Etapa 1.-



A continuación, pasamos a describir el algoritmo, de forma más detallada.

```
MIN-MAX (<0,NI>)
   AB := {<0,NI>}                               ;ver nota 1 posterior
   CE := Ø
   CAMINO := Ø
   NE := Ø
   CICLO-GENERACION.

CICLO-GENERACION ()
   SI  AB = Ø  ENTONCES  CICLO-EVALUACION
   CASO CONTRARIO:
   pn := un elemento cualquiera de AB       ;id. nota 2
   AB := AB - {pn}
   CE := CE U {pn}
   SI  prof = cota  ENTONCES  CICLO-GENERACION  ;id. nota 3
   CASO CONTRARIO:                              ;id. nota 4
      sucs := sucesores de pn mediante S''
      Para cada <1+prof,nod'> ∈ sucs :
         AB := AB U {<1+prof,nod'>}
         CAMINO := CAMINO U {<pn, <1+prof,nod'>>}
   CICLO-GENERACION.

CICLO-EVALUACION ()
   Ordenar los elementos de CE según profundidad ;id. nota 5
   Para cada pn ∈ CE :
      SI  prof = cota  ∨  ¬∃<pn,pn'> ∈ CAMINO
          ENTONCES  EV(pn) := h(pn)
                    NE := NE U {<h(pn),pn>}
          CASO CONTRARIO:
             SI  prof = par
                 ENTONCES  EV(pn) := máx {EV(pn') : <pn,pn'>}
                           NE := NE U {<EV(pn),pn>}
                 CASO CONTRARIO
                           EV(pn) := min {EV(pn') : <pn,pn'>}
                           NE := NE U {<EV(pn),pn>}
   Elegir nod tal que: <<0,NI>,<1,nod>> ∈ CAMINO  ∧
                      EV(<1,nod>) = máx {EV(<1,nod'>) :
                                    <<0,NI>,<1,nod'>> ∈ CAMINO}.
```

1.2.1 *Notas.-*

Nota 1.- Los datos de entrada al algoritmo son NI, COTA, h. Consideraremos <profundidad, nodo> en los elementos de los conjuntos AB y CE. Estos elementos se llamarán pnodos. La profundidad de NI es cero.

AB representa el conjunto de pnodos que no han sido expandidos, CE representa el conjunto de pnodos expandidos, CAMINO representa el conjunto de arcos del árbol generado, y NE representa el conjunto de pnodos evaluados. Ahora al hablar de arcos puede parecer que entramos en inconsistencia con lo que hemos definido como árbol de juego ó árbol alternado (definiciones II-3.3 y 3.7), ya que allí hablábamos de niveles alternados de 1-arcos y de k-arcos. Esto no es así pues, ahora sólo vamos a tratar con 1-arcos de un nodo a cada uno de sus sucesores, ya que, el carácter de 1-arco ó de k-arco se va a reflejar según estemos en un nivel maximizante ó en un nivel minimizante. Por otra parte el problema que puede representar el tener que pasar de k-arcos a 1-arcos queda reflejado en la función que calcula los sucesores de un nodo dado, y esta función como hemos dicho depende de cada juego. Con estas consideraciones que hemos expresado, podemos suponer que nuestro algoritmo trata con 1-arcos solamente. Nótese también que la función que calcula los sucesores de un nodo dado debe también calcular su profundidad.

Nota 2.- Tomamos el primer elemento de AB por facilidad de programación. La ordenación de AB es lo de menos, pues hemos de desarrollar el árbol completo hasta una profundidad igual a COTA.

Nota 3.- Prof(n) representa una función que nos da la profundidad de un pnodo, es decir, su primer componente.

Nota 4.- SUCESORES(n) será una función particular de cada juego. La variable sucs recoge el conjunto de sucesores de un nodo dado. Con la notación de conjuntos que venimos empleando se tiene, SUCESORES(n) = $\{m \in E'' : nS''m\}$. Recuérdese que el algoritmo hace que los nodos se consideren junto con su profundidad, lo cual no representa más dificultad que adaptar el indicador de jugador en el árbol de juego J(s,cota), subgrafo de <E'',S''>. Podemos considerar que en vez de los valores 0 y 1, este indicador toma valores enteros positivos y que según la paridad de este número sabemos a que jugador pertenece el nodo. Es decir, en vez de tener

$E = N \times \{0,1\}$, tenemos $E = N \times \mathbb{N}$, en vez de $T = \{\langle n,0\rangle : n \in N\}$, tenemos $T = \{\langle n,k\rangle : n \in N \wedge [k \in \mathbb{N} : k \text{ par}]\}$, donde $\langle N,R'\rangle$ y $F \subseteq N$ vienen dados por ser $\langle E,R,T,G,P\rangle$ un juego interpretable en grafos. Como se podrá comprobar más adelante, este hecho no presenta mayores problemas de programación.

Nota 5.- La ordenación de CE es debida a razones de operatividad del algoritmo.

1.3.- Programa.

Damos ahora la sintaxis en lenguaje Lisp, de un programa que implementa el procedimiento Min-Max. Nótese que este módulo de programación es incompleto, pues le falta la programación de los aspectos particulares relativos al juego en cuestión.

```
;Programación del Procedimiento Min-Max.
;
(de  JUGADA  (nod)
   (cadr  (cadar  (sort  'ORDEN
                   (mapcar  '(lambda (x) (list (EV  x) x))
                             (SUCES  (list 0 nod))  ))  ))  )
;
(de  EV  (pnodo)
   (let ((p (PROF  pnodo))  (sucs (SUCES  pnodo)))
        (cond ((or (= p cota) (null sucs)) (H  pnodo))
              ((eveno p) (PMAX  sucs))
              (t (PMIN  sucs))  ))  )
;
(de  PMAX  (lista)
   (apply 'max (mapcar 'EV  lista))  )
;
(de  PMIN  (lista)
    (apply 'min (mapcar 'EV  lista))  )
;
(de  ORDEN  (a b)
   (> (car a) (car b))  )
;
(de  PROF  (pnodo)
   (car pnodo)  )
```

;Documentación:

;Funciones	Variables Locales	Variables Globales
;JUGADA	nod, x	cota
;EV	pnodo, p, sucs	
;PMAX	lista	
;PMIN	lista	
;ORDEN	a, b	
;PROF	pnodo	

1.3.1.- *Notas.*

Como ya hemos anticipado, el programa anterior no es completo. De momento para nosotros, SUCES significará la función que calcula los sucesores de un pnodo dado, para un juego dado. De forma análoga, H será la función de evaluación estática usada para evaluar pnodos con profundidad que alcance la cota, ó para pnodos sin sucesores. Recuérdese que se trabajaba con elementos, <profundidad, nodo>, en vez de con nodos. Esta nueva entidad formada por la profundidad y el nodo, es lo que en el programa se llama pnodo.

Respecto a las variables que aparecen en el programa, diremos que representan las distintas entidades con las que trabaja el programa; por otra parte, estas entidades son también las que maneja el algoritmo dado anteriormente en 1.2. De forma explícita, estas entidades son el nodo, el pnodo y un número entero. Las variables del programa toman como valor una de las entidades anteriores ó listas con agrupaciones de las entidades anteriores. Pasamos a continuación a comentar someramente las funciones integrantes del programa.

1.3.2.- JUGADA.

Toma como argumento un nodo del juego ($\in$ E), que hace el papel de nodo raíz ó nodo inicial, y devuelve el sucesor con mayor evaluación. Para ello, transforma el nodo raíz en pnodo raíz, calcula sus sucesores, los evalúa, los ordena según la evaluación, y elige el de mayor evaluación.

1.3.3.- EV, PMAX, PMIN.

EV realiza la evaluación de los nodos. Para nodos con la

cota de profundidad alcanzada ó sin sucesores, les aplica la función H. Para el resto de nodos, realiza la propagación de las evaluaciones según propone el procedimiento Min-Max. Pa.a realizar esta propagación se ayuda de las funciones auxiliares PMAX y PMIN, que devuelven el máximo y mínimo respectivamente, de la lista que se obtiene de aplicar EV a cada elemento de una lista de pnodos.

1.3.4.- ORDEN, PROF.

La primera de ellas, se utiliza para ordenar una lista de pnodos según la evaluación. La segunda nos da la profundidad de un pnodo dado.

Pasamos en el siguiente punto a aplicar el procedimiento Min-Max a un juego concreto. Desde el punto de vista de la programación esto supone la codificación de las funciones SUCES y H, en el programa anterior. Antes de realizar esta tarea, daremos la descripción del juego elegido para la aplicación.

1.4.- Descripción del Juego.

1.4.1.- *Introducción.*

De todos es conocido el juego del "tres en raya" (trincarro ó tic-tac-toe). Este juego no se adapta muy bien a nuestros objetivos, pues en primer lugar, el árbol de su desarrollo es bastante pequeño (cuando el interes de este procedimiento es para árboles grandes), y en segundo lugar, no se trata propiamente de un problema de inteligencia artificial. En efecto, una de las características especificas de este campo es la ausencia de algoritmos que solucionen el problema de forma determinista; pues bien, en el caso del tres en raya el juego está absolutamente determinado desde el comienzo del mismo. Existen dos versiones fundamentales de este juego, con ó sin deslizamiento.

El motivo de haber hablado del conocido tres en raya, es que las reglas para colocar y mover las piezas son exactamente iguales, a las del juego al que se va a aplicar el procedimiento Min-Max. Este juego, que por analogía al tres en raya vamos a llamar Tres en Raya Generalizado con Deslizamiento (TRGD), se juega con tres piezas por jugador, y la diferencia con el tres en raya "convencional" está en el tablero. Esto implica, como es

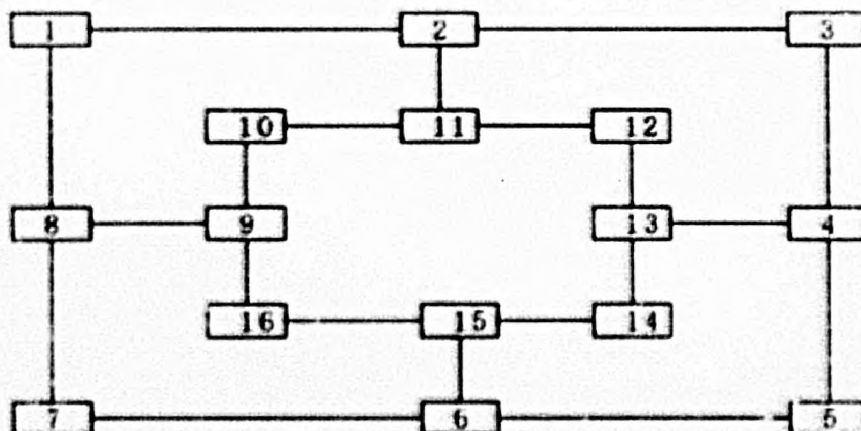
natural, diferencias en cuanto a las posiciones ganadoras y perdedoras entre ambos juegos. Hemos optado por este juego, en primer lugar por la simplicidad de su descripción, y en segundo porque subsana la pega que tenía tres en raya, al no haber, al menos que conozcamos, una estrategia determinista que gane el juego. Por otra parte, el árbol del juego correspondiente también es considerablemente más grande (con más estados y con más movimientos) que el del tres en raya convencional.

La elección de este juego también esta motivada, además de su no trivialidad, por el hecho de que, si bien la gama de juegos posibles es amplia, no ocurre lo mismo con las funciones de evaluación estática disponibles para dicha gama de juegos.

No obstante, no nos preocupa mucho la elección del juego, ni incluso la de una buena función de evaluación estática, ya que el objetivo del juego es constatar que los programas funcionan, siendo nuestro interés la aplicación de estos procedimientos a la dedución automática. En otras palabras, el objetivo de nuestra memoria es el de aplicar los procedimientos Min-Max y Alfa-Beta a la deducción automática y no el de programar juegos ó buscar buenas funciones de evaluación estática para juegos. Sin más, pasamos a describir el juego del Tres en Raya Generalizado con Deslizamiento.

1.4.2.- *Tres en Raya Generalizado con Deslizamiento.*

El juego lo juegan dos jugadores, cada uno de ellos con tres fichas iguales, pero diferentes de las del adversario, con el siguiente tablero:



donde hemos numerado cada una de las posiciones del tablero, según se muestra.

Se juega alternativamente. Cada jugada consiste en colocar una ficha propia en una posición libre del tablero (si es que no se han colocado las tres), ó deslizar una ficha propia desde su posición a una posición contigua y libre del tablero.

Gana el juego el jugador que coloque primero sus tres fichas en alguna de las ocho "líneas" con tres posiciones que contiene el tablero. Estas "líneas", las vamos a representar por una terna que contenga las 3 posiciones que las definen. Así, tendremos, que el conjunto de líneas = $\{\langle 1,2,3\rangle, \langle 3,4,5\rangle\ \langle 5,6,7\rangle, \langle 1,7,8\rangle, \langle 10,11,12\rangle, \langle 12,13,14\rangle, \langle 14,15,16\rangle, \langle 9,10,16\rangle\}$.

El estado ó nodo inicial del juego es el tablero vacío. Los nodos se representarán por dos ternas, que indican las posiciones de tablero ocupadas por ambos jugadores. Así, el nodo inicial es $\langle\langle\rangle\ \langle\rangle\rangle$. Para poner una ficha, la colocamos en cualquier posición libre del tablero.

Finalmente, por necesidades impuestas por el procedimiento Min-Max, consideramos en vez de los nodos descritos tres párrafos antes, los pnodos, que están compuestos por un nodo y su profundidad, en el orden: pnodo = $\langle$profundidad, nodo$\rangle$, como hemos dicho.

En principio, el juego sería la 5-tupla siguiente, $\langle E,R,T,G,P\rangle$, donde:

$E = \{\langle\langle n_1,n_2\rangle,i\rangle$: $n_1,n_2 = \langle p,q,r\rangle$, $1 \le p,q,r \le 16$, todos distintos, con algunos p,q,r iguales a blancos ó todos blancos para el nodo inicial, $i = 0,1$ según juege Max ó Min$\}$.

R está definida de la siguiente forma, sean $n,m \in E$, con

$$n = \langle\langle\langle p,q,r\rangle,\langle p',q',r'\rangle\rangle,i\rangle \quad \text{y}$$

$$m = \langle\langle\langle s,t,u\rangle,\langle s',t',u'\rangle\rangle,j\rangle, \quad \text{entonces,}$$

$$nRm \iff i \neq j$$
$$\wedge\ \{\ [\langle p,q,r\rangle R_0 \langle s,t,u\rangle \wedge \langle p',q',r'\rangle = \langle s',t',u'\rangle]$$
$$\vee\ [\langle p',q',r'\rangle R_0 \langle s',t',u'\rangle \wedge \langle p,q,r\rangle = \langle s,t,u\rangle]\ \}$$

con R_0, $\langle p,q,r\rangle M\langle s,t,u\rangle \leftrightarrow$

$\langle s,t,u\rangle \in PONE(\langle p,q,r\rangle)$ ó $\langle s,t,u\rangle \in DESLIZA(\langle p,q,r\rangle)$.

El definir está relación con notación de conjuntos, hace que la definición sea engorrosa. Por eso se han utilizado las funciones PONE y DESLIZA, que serán más tarde definidas en el programa. La primera se utiliza para la primera parte del juego, y como su nombre indica, se utiliza para calcular sucesores que se obtienen poniendo fichas en el tablero. La segunda, calcular los sucesores que se obtienen deslizando fichas en el tablero.

$T = \{\langle\langle n_1,n_2\rangle,0\rangle \in E\}$

$G = \{\langle\langle n_1,n_2\rangle,1\rangle : n_1 \in$ conjunto de líneas$\}$

$P = \{\langle\langle n_1,n_2\rangle,0\rangle : n_2 \in$ conjunto de líneas$\}$

El conjunto de líneas se definió explícitamente al comienzo de este punto. Una línea es una terna de posiciones en un mismo segmento de los ocho de que se compone el tablero. Si una línea está ocupada por fichas del mismo jugador, se dice entonces que la línea es ganadora para el jugador dueño de las fichas.

Hemos mencionado antes que el carácter blanco (ausencia de carácter) es también utilizado en las ternas que definen los estados del juego, para significar que todavía no hemos colocado todas nuestras fichas.

Veamos pues, que éste juego es interpretable en grafos. Para ello hemos de encontrar el grafo del juego $\langle N,R'\rangle$, y el subconjunto $F \subseteq N$, que satisfagan la definición de juego interpretable en grafos dada en 1-2.25.

En efecto, consideremos los siguientes conjuntos y la siguiente relación binaria,

$N = \{\langle n_1,n_2\rangle : n_1 = \langle p,q,r\rangle,\ n_2 = \langle p',q',r'\rangle,\ 1 \le p,q,r \le 16,$
$1 \le p',q',r' \le 16$ todos distintos, con algunos p,q,r iguales a blancos ó todos blancos para el nodo inicial$\}$.

R' está definida como sigue: Sean $n,m \in N$, decimos que

$nR'm \leftrightarrow \langle n,0\rangle R\langle m,1\rangle$ ó $\langle n,1\rangle R\langle m,0\rangle$.

$F = \{\langle n_1,n_2\rangle \in N :$ con n_1 ó $n_2 \in$ conjunto de líneas$\}$.

Trivialmente, se puede comprobar que el grafo $\langle N,R' \rangle$ y el subconjunto $F \subseteq N$, verifican la definición de juego interpretable en grafos para el juego inicial $\langle E,R,T,G,P \rangle$.

No obstante, de cara al programa que daremos en el siguiente apartado, no vamos a utilizar esta representación. De hecho ya lo hemos dicho cuando esbozamos la representación del juego al comienzo de esta descripción. Esta anomalía se justifica por lo siguiente. Nosotros para distinguir a qué jugador pertenece un nodo no vamos a utilizar el valor 0 ó 1 en la variable que a tal efecto figura en la representación del nodo, sino que vamos a utilizar para dicho fin, el carácter par ó impar de los números enteros positivos.

Es decir, se tratará con un juego isomorfo al inicial $\langle E,R,T,G,P \rangle$, con

$E_1 = \{\langle\langle n_1,n_2\rangle,j\rangle : n_1,n_2 = \langle p,q,r\rangle,\ 1 \leq p,q,r \leq 16,$

todos distintos con algunos p,q,r, iguales a blancos o todos blancos para el nodo inicial, j = p,i reflejando caracter par ó impar de la profundidad del nodo$\}$

Es trivial, que $f: E \longrightarrow E_1$ definida por: dado $n \in E$,

$$f(n) = \begin{cases} \langle n_1,n_2,p\rangle & \text{si} \quad n = \langle n_1,n_2,0\rangle \\ \langle n_1,n_2,i\rangle & \text{si} \quad n = \langle n_1,n_2,1\rangle \end{cases}$$

es un isomorfismo entre juegos.

Además, tampoco es ésta la representación definitiva del juego que utilizamos en el programa. Por exigencias de Min-Max, cada nodo debe de ir arrastrando su profundidad, de ahí que cada nodo lo daremos junto con su profundidad, en vez de expresando solamente el carácter par ó impar de ésta.

Con estas consideraciones, justificamos la representación para nodos dada al principio.

Sobre este juego, de cara a la aplicación del procedimiento Min-Max, nos falta dar la función de evaluación estática que utilizaremos. Tomaremos $h: N \longrightarrow \mathbb{Z}$, definida de la siguiente forma: sea $n \in N$, $n = \langle\langle p,q,r\rangle,\langle p',q',r'\rangle\rangle$; definimos $h(n)$ como el número de líneas abiertas para la terna $\langle p,q,r\rangle$, menos el número de líneas abiertas para $\langle p',q',r'\rangle$. $\mathbb{Z}$ representa el conjunto de los números enteros.

Aparece un nuevo concepto, el de "línea abierta", que pasamos a matizar. En este mismo apartado, hemos explicado el concepto de línea y de línea ganadora cuando se dió el tablero del juego. Una línea está abierta para un jugador, si sólo contiene blancos (posiciones libres) ó fichas de dicho jugador. Nótese que esta función está dentro de las condiciones dadas en la definición de función de evaluación estática de 1.1.2 y precisada en 1.1.3.

Nótese que para cada nodo $n = \langle\langle p,q,r\rangle,\langle p',q',r'\rangle\rangle$, la relación que existe entre el valor de la evaluación estática, según forme parte de un pnodo MAX ó un pnodo MIN, es la siguiente:

Sea $m_1 = \langle p,n\rangle$ un nodo MAX, $m_2 = \langle q,n\rangle$ un nodo MIN, se tiene, $h(m_1) = -h(m_2)$ $\forall n \in N$, con $\langle N,R'\rangle$ correspondiente a la definición de juego interpretable en grafos. Nótese que ésto es debido a que en realidad m_1 y m_2 son el mismo nodo, pero en el primer caso el nodo es MAX y en el segundo caso es MIN.

1.5.- Aplicación del procedimiento Min-Max al juego del Tres en Raya Generalizado con Deslizamiento.

Retomamos ahora la sintaxis del programa Lisp, que implementaba el procedimiento Min-Max, dado en el apartado 1.3. Nos planteamos escribir las funciones que implementan el juego y la función de evaluación estática que usaba.

1.5.1.- *Notas.*

Antes de escribir estos programas, recordamos también las notas 1.2.1 y 1.3.1. Dábamos allí las entidades con las que iba a trabajar en programa. Al tener ahora definido el juego al que vamos a aplicar el procedimiento Min-Max, podemos dar estas entidades en sintaxis Lisp. Así, se tiene,

nodo = ((p q r) (p'q'r'))

pnodo = (profundidad ((p q r) (p'q'r')))

número entero = representa en algunos casos la profundidad y en otros valores de la evaluación de nodos.

De esta forma podemos dar el siguiente cuadro, que refleja las variables del programa dado en 1.3, y los valores que toman estas variables.

Variable	Valor
nod	nodo
x	pnodo
pnodo	pnodo
p	profundidad ($\in \mathbb{N}$)
sucs	lista (Lisp) de pnodos
lista	lista (Lisp) de números
a, b	(evaluación pnodo)
cota	número entero

1.5.2.- *Programa.*

Damos ahora la codificación en Lisp, de las funciones que implementan el juego anterior. Estas funciones junto con las que figuran en el programa 1.3, constituyen el programa de aplicación del procedimiento Min-Max al juego dado en el apartado anterior 1.4.

Como en el caso precedente dado en el apartado 1.3, daremos primero la codificación completa y a continuación comentaremos las funciones y variables que integran el programa.

```
;Programación de las funciones de evaluación estática y de
;cálculo de sucesores para el juego del T.R.G.D.
;Asignación a variables globales
;
(setq  cota  5  ni  '(() ()))

(setq  lineas  '( (1 2 3) (3 4 5) (5 6 7) (1 7 8) (9 10 16)
                  (10 11 12) (12 13 14) (14 15 16) ))

(plist  'vecinos  '( 1 (2 8) 2 (1 3 11) 3 (2 4) 4 (3 5 13) 5 (4 6)
                     6 (5 7 15) 7 (6 8) 8 (7 9 1) 9 (8 10 16)
                     10 (9 11) 11 (10 12 2) 12 (11 13)
                     13 (12 14 4) 14 (13 15) 15 (14 16 6)
                     16 (9 15) ) )

;
```

```
;1.- Función de evaluación estática  H
;1.a.- Funciones principales

(de  H  (pnodo)
   (let  ((p (PROF  pnodo)) (pos+  (car (NODO  pnodo)))
                             (pos-  (cadr (NODO  pnodo))) )
      (* (if (evenp p) 1 -1)
             (apply  '+  (mapcar  'HAUX  lineas)) )) )
;
(de  HAUX  (linea)
   (let  ((aa (INTER pos+ linea)) (bb (INTER pos- linea)))
      (cond  ((equal pos+ linea) 100)
             ((equal pos- linea) -100)
             ((and (null aa) (null bb)) 0)
             ((null bb) 1)
             ((null aa) -1)
             (t 0)   ))  )

;1.b.- Funciones auxiliares
;Además de la función  PROF ya dada en el programa 1.3

(de  INTER  (x y)
   (cond  ((or (null x) (null y)) '() )
          ((member (car x) y)
                   (cons (car x) (INTER (cdr x) y)))
          (t (INTER  (cdr x) y))   )  )


;2.- Función de cálculo de sucesores
;2.a.- Funciones principales
 (de  SUCES  (pnodo)
    (let  ((p (PROF  pnodo)) (nod (nodo pnodo)))
      (let  ((n+ (NUM+  nod)) (n- (NUM-  nod)))
        (cond ((= n+ 0) '( (1 ((1) ())) (1 ((2) ()))
                           (1 ((9) ())) (1 ((10) ())) ) )
              ((> n+ n-) (mapcar '(lambda (x) (list (+ 1 p) x))
                                 (PONE '- nod) ) )
              ((< n+ 3)  (mapcar '(lambda (x) (list (+ 1 p) x))
                                 (PONE '+ nod) ) )
              (t (mapcar '(lambda (x) (list (+ 1 p) x))
                 (DESLIZA  (if (evenp p) '+ '-) nod ) ) ) ) )))
```

```
(de  PONE  (jug nod)
    (cond  ((equal jug '+)
               (let  ((libres
                        (DIF '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
                              (append (car nod) (cadr nod)) )) )
                  (mapcar '(lambda (x) (list (METE x (car nod))
                                       (cadr nod) ))  libres) ))
           (t  (let  ((libres
                        (DIF '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
                             (append (car nod) (cadr nod)) )) )
                   (mapcar '(lambda (x) (list (car nod)
                              (METE x (cadr nod))))  libres)))))
;
(de  DESLIZA  (jug nod)
         (let ((ocupados (append (car nod) (cadr nod))) )
            (if (equal jug '+)  (apply 'append
               (mapcar '(lambda (x)  (mapcar '(lambda (y)
                 (list (METE y (remove x (car nod))) (cadr nod)))
                    (DIF (get 'vecinos x) ocupados))) (car nod)))
                 (apply 'append  (mapcar '(lambda (x)
                    (mapcar '(lambda (y)  (list (car nod)
                            (METE y (remove x (cadr nod)))) )
                              (DIF (get 'vecinos x) ocupados) ) )
                                (cadr nod))))))

;2.b.- Funciones auxiliares

(de  METE  (a l)
   (cond  ((null l) (list a))
          ((= (length l) 1) (if (< a (car l)) (cons a l)
                                       (list (car l) a) ))
          (t  (let  ((b (car l)) (c (cadr l)))
                 (cond  ((< a b) (list a b c))
                        ((< a c) (list b a c))
                        (t (list b c a)) )) )) )
;
(de  DIF  (x y)
   (cond  ((null x) ())
          ((member (car x) y) (DIF (cdr x) y))
          (t (cons (car x) (DIF (cdr x) y))) ))
```

```
;3.- Funciones de acceso

(de  NODO  (pnodo)
   (cadr pnodo))
;
(de  NUM+  (nod)
   (length (car nod)))
;
(de  NUM-  (nod)
   (length (cadr nod)) )
```

;documentación

;	Funciones	Variables Locales	Variables Globales
;	H	pnodo, p, pos+, pos-,	lineas
;	HAUX	linea, aa, bb, pos+,	
;		pos-,	
;	INTER	x, y,	
;	SUCES	pnodo, p, nod, n+, n-,	
;	PONE	jug, nod, libres	
;	DESLIZA	jug, nod, ocupados,	vecinos
;	METE	a, l, b, c,	
;	DIF	x, y,	
;	NODO	pnodo	
;	NUM+	nod	
;	NUM-	nod	

Damos un breve comentario de las funciones y variables, usadas en la codificación anterior.

1.5.3.- H, HAUX, INTER.

Implementan la función de evaluación estática que se usa en el juego. La función H, toma un pnodo, calcula las posiciones ocupadas por MAX y por MIN, y según la profundidad (juega MAX ó MIN) toma como factor ±1, para calcular el valor de la evaluación estática para un pnodo. Este valor es el resultado de ir contando el número de líneas abiertas para cada jugador, restar estos números y multiplicar por el factor ±1 según se trate de MAX (+1) ó de MIN (-1). El cómputo de líneas abiertas para ambos jugadores lo realiza la función HAUX. Finalmente, la función auxiliar INTER, implementa la intersección de conjuntos (los conjuntos en lisp los representaremos por listas).

1.5.4.- SUCES, PONE, DESLIZA, METE, DIF.

Implementan el cálculo de los sucesores de un pnodo dado. La función de más alto nivel es SUCES, que discrimina entre realizar un movimiento inicial, poner MAX, poner MIN, deslizar MAX ó deslizar MIN. La función PONE realiza la primera parte del juego, para ambos jugadores, colocando nuevas fichas en el tablero. La función DESLIZA, realiza la segunda parte de juego, para ambos jugadores, realizando deslizamientos legales de fichas. Ambas funciones toman como argumentos un nodo y una variable que especifica a un jugador. Ambas también, utilizan las funciones auxiliares METE y DIF. En cuanto a estas últimas, la primera de ellas introduce una ficha en el tablero, y la segunda realiza la diferencia entre dos conjuntos.

1.5.5.- NODO, NUM+, NUM-.

La primera de ellas toma como argumento un pnodo y devuelve el nodo correspondiente. NUM+ y NUM-, toman como argumento un nodo y calculan el número de posiciones de MAX y el número de posiciones de MIN respectivamente.

1.5.6.- *Calificación de variables.*

Como ya hicimos en 1.5.1, damos la relación de variables junto con los valores que toman. Omitimos las variables ya calificadas en dicho apartado 1.5.1.

<u>Variable</u>	<u>Valor</u>
pos+	<p,q,r> posición MAX
pos-	<p',q',r'> posición MIN
linea	<1,2,3>,...,<9,10,16>
aa, bb	<>, <a>, <a,b>, <a,b,c>
x, y	listas de Lisp
n+	número de + de un nodo
n-	número de - de un nodo
jug	+ ó -
a	número de 1 al 16
l	lista de 1, 2 ó 3 números del 1 al 16
b, c	números del 1 al 16
ocupados	lista de números del 1 a 16

vecinos	(1 (2 8),...,16 (9 15))
libres	lista de números del 1 a 16

Nota.- Un número del 1 al 16 representa una posición de tablero.

5.2.- PROCEDIMIENTO ALFA-BETA.

2.1.- Descripción del procedimiento.

2.1.1.- *Notas*.

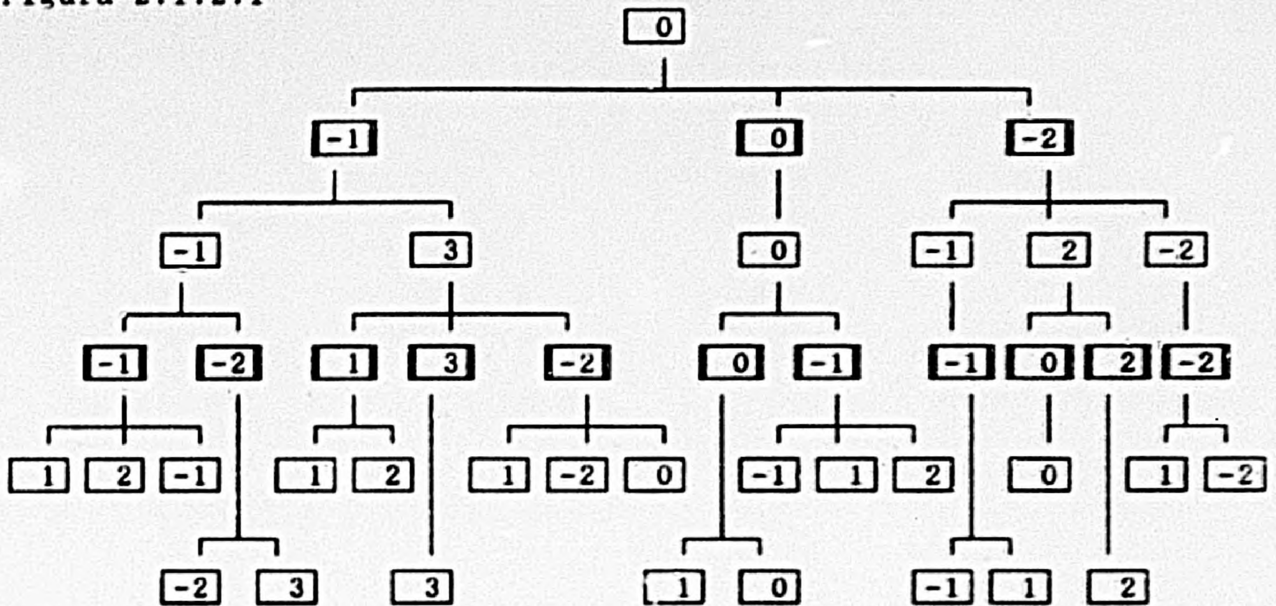
Para este procedimiento, aceptaremos las hipótesis de partida del procedimiento Min-Max; es decir, se parte de juegos interpretables en grafos. Se pretende buscar estrategias (próximo movimiento) para el jugador que comienza el juego. Ante la imposibilidad para numerosos juegos del desarrollo exhaustivo del árbol de juego, hemos de emplear técnicas que de alguna manera acorten este desarrollo y den buenos movimientos. Al haber acotación, se pierde la posible certeza, pero para numerosos juegos, como se ha dicho, no tenemos otra posibilidad.

El procedimiento Alfa-Beta está considerado como un refinamiento del procedimiento Min-Max, que ofrece la misma elección de movimiento, pero con un coste menor de computación, debido a que rechaza el desarrollo y evaluación de caminos de peor evaluación. Esto quiere decir que no se mantiene la separación entre el proceso de desarrollo del árbol y el proceso de evaluación, como ocurría en el procedimiento Min-Max. Esto va a ser causa de que la descripción algorítmica del procedimiento varíe de la dada para el procedimiento Min-Max. Por otra parte, tampoco realiza el procedimiento Alfa-Beta el desarrollo total del árbol del juego. Estas mejoras se llevan a cabo mediante el mantenimiento dinámico de una cotas a los valores de evaluación de los nodos MAX y de los nodos MIN.

2.1.2.- *Ejemplo*.

El siguiente gráfico representa un árbol de juego generado por el procedimiento Min-Max. También se representan las evaluaciones estáticas de los nodos de último nivel, la propagación de dichas evaluaciones y en consecuencia la elección de jugada. Este ejemplo se utilizará para ilustrar el procedimiento Alfa-Beta. También servirá para notar los "cortes" que realiza el procedimiento Alfa-Beta en el árbol de juego generado por el procedimiento Min-Max.

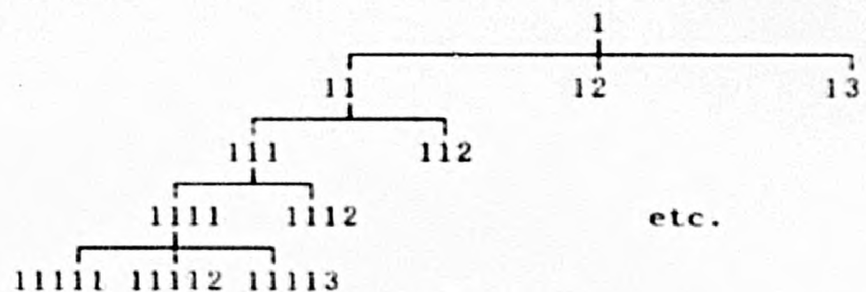
Figura 2.1.2.1



Con ayuda del ejemplo dado, daremos primero los conceptos referentes al procedimiento Alfa-Beta. Luego daremos el árbol resultante del procedimiento Alfa-Beta para apreciar la diferencia con el anterior generado por el Min-Max.

Para la identificación de cada nodo, utilizaremos la siguiente numeración:

Figura 2.1.2.2.



2.1.3.- *Procedimiento Alfa-Beta.*

En el ejemplo dado en 2.1.2, al calcular el valor de evaluación del primer descendiente del nodo 112, tenemos que dicho valor es 1. Esto da un valor para el nodo 112 mayor ó igual que 1 puesto que el valor del nodo 112 es el máximo del de sus sucesores. Como el valor del nodo 111 es -1, quiere decir que podemos asignar al nodo 11 un valor de -1, sin necesidad de explorar el resto de caminos que parten de 112, ya que no van a modificar la evaluación del nodo 11 debido a la propia mecánica del procedimiento Min-Max (el valor del nodo 11 es el mínimo de

los valores de sus sucesores). El procedimiento Alfa-Beta aprovecha esta idea, no realizando la generación ni evaluación de los caminos rechazados.

2.1.3.1.- *Definición.*

Sea <E,R,T,G,P> un juego interpretable en grafos, sea J(nodo) el árbol de juego para un determinado nodo inicial.

Llamamos *valor-α* de un nodo MAX, y lo denotamos por V_α(nodo) al valor máximo de las evaluaciones de sus sucesores. De forma análoga, llamamos *valor-β* de un nodo MIN, V_β(nodo), al valor mínimo de las evaluaciones de sus sucesores.

2.1.3.2.- *Notas.*

Al avanzar en la generación ó desarrollo del árbol de juego, el valor-α de un nodo MAX nunca decrece, y el valor-β de un nodo MIN nunca crece.

La diferenciación entre V_α , aplicable a un nodo MAX, y V_β , aplicable a un nodo MIN, hace necesario diferenciar el jugador en cada estado del juego. Esto lo haremos, como en el procedimiento Min-Max, arrastrando la profundidad de cada nodo.

2.1.3.3.- *Reglas de Corte.*

El procedimiento Alfa-Beta, parte de las mismas condiciones y opera de forma análoga al procedimiento Min-Max, salvo en la siguiente diferencia que consiste en detener el proceso de generación y evaluación de nodos para los casos de:

R.1.- Nodos MAX cuyo valor V_α sea mayor ó igual al valor V_β de su antecesor MIN, y como valor de evaluación del nodo MAX se toma su valor V_α.

R.2.- Nodos MIN cuyo valor V_β sea menor ó igual al valor V_α de su antecesor MAX, y como valor de evaluación del nodo MIN se toma su valor V_β.

Cuando se desecha un nodo (y los caminos que parten de él) por la regla R.1 (respectivamente por R.2), se dice que se ha hecho un α-corte (respectivamente β-corte). El procedimiento

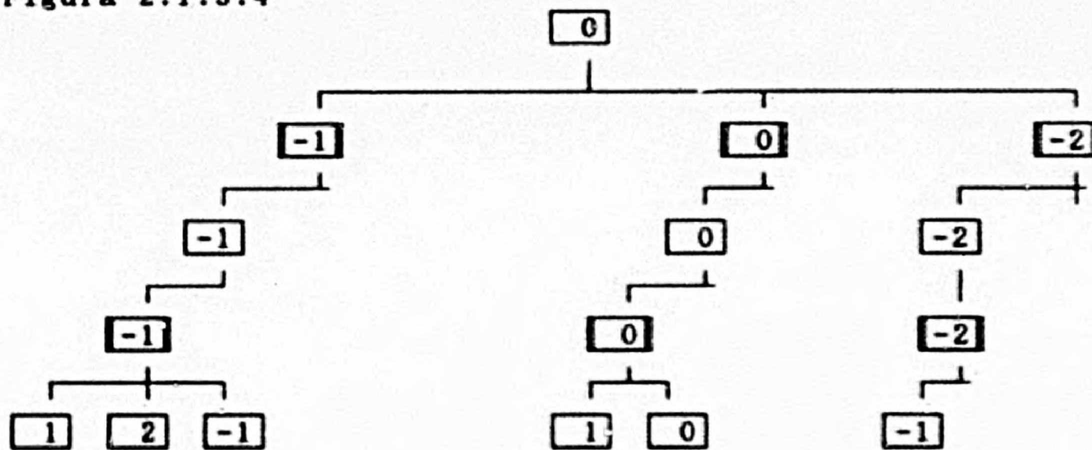
Alfa-Beta, es así, un procedimiento de "poda" del árbol. Es evidente que estos cortes no afectan en nada a la eficiencia del procedimiento Min-Max, al cual mejoran.

2.1.3.4.- *Notas.-*

De la definición anterior se deduce que, al ir calculando valores V_α y V_β , hemos de tener presente un valor de referencia sobre la evaluación del antecesor, que actúe como cota y permita rechazar los nodos según establecen R.1 y R.2 del criterio anterior. Este hecho se plasmará (en el algoritmo y programa correspondientes), haciendo depender dichos valores V_α y V_β no solamente del nodo en cuestión sino además de otro parámetro que represente el valor provisional de la evaluación de su antecesor.

A continuación, vamos a dar el árbol de juego equivalente al árbol dado en la figura 2.1.2.1, aplicando el procedimiento Alfa-Beta en vez del procedimiento Min-Max.

Figura 2.1.3.4



2.2.- Algoritmo.

A primera vista, el algoritmo Alfa-Beta debe ser parecido al Min-Max salvo en la etapa de evaluación de nodos, debido a los cambios que originan las reglas de corte dadas en 2.1.3.3 (recuérdense las etapas del procedimiento Min-Max, dadas al comienzo del apartado 1.2). Además la generación del árbol debe ir unida a su evaluación, para poder efectuar los cortes. Debido a estas diferencias, la forma del algoritmo Alfa-Beta varía considerablemente con respecto a la forma del algoritmo Min-Max.

Como en el caso del algoritmo Min-Max, daremos este algoritmo en dos formas, una más general y otra más detallada. Después, comentaremos brevemente el algoritmo.

Forma general del algoritmo Alfa-Beta

Elección de jugada

1.- Tomar NI, y asignar profundidad 0.

2.- Calcular sucesores de <NI,0>.

3.- Evaluar sucesores de <NI,0>.

4.- Elegir mejor sucesor.

Evaluación y poda del árbol

5.- Tomar el nodo a evaluar y su profundidad.

6.- Si el nodo es MAX, calcular V_α del nodo
si no (nodo MIN), calcular V_β del nodo.

Valor-α

7.- Si el nodo es un objetivo MIN, dar $V_\alpha := -\infty$
si no:
8.- Si el nodo no tiene sucesores ó se alcanza la cota de profundidad, dar $V_\alpha :=$ H(nodo)
si no:
9.- Para cada sucesor del nodo:
Calcular V_β del sucesor.
Si $V_\beta \leq V_\alpha$ provisional del padre: corte-β
si no, tomar como V_α del nodo el V_β del sucesor.

Valor-β

10.- Si el nodo es un objetivo MAX, dar $V_\beta := +\infty$
si no:
11.- Si el nodo no tiene sucesores ó se alcanza la cota de profundidad, dar $V_\beta :=$ H(nodo)
si no:
12.- Para cada sucesor del nodo:
Calcular el V_α del sucesor.
Si $V_\alpha \geq V_\beta$ provisional del padre: corte-α
si no, tomar como V_β del nodo el V_α del sucesor.

Calculo de sucesores.

13.- Tomar pnodo a expandir.

14.- Calcular sucesores del nodo y sus profundidades.

Forma detallada del algoritmo Alfa-Beta.

ALFA-BETA (<0,NI>)

EV := Ø [[AB = Ø]]

sucs := SUCES(<0,NI>)

V_{ap} := Ø

V_{bp} := Ø

Para cada m ∈ sucs :

mm := <m,VALOR-B(m,V_{ap})>

EV := EV ∪ {mm}

Elegir nodo correspondiente a mm ∈ EV tal que:

$valor\text{-}b_{mm}$ = máx. {valor-b : <k,valor-b> ∈ EV}

VALOR-A (n,V_{bp})

sucs := SUCES(n) [[AB := AB ∪ sucs; sucs-aux := sucs]]

SI [sucs=Ø ∨ Prof(n)=cota], ENTONCES, valor-a := H(n)

SI NO: V_p := Ø

Hasta que sucs = Ø:

ns := primer elemento de sucs

sucs := sucs - {ns}

val-b := VALOR-B(ns,V_p)

SI val-b > V_p : V_p := val-b.

[[AB = AB-{ns}]]

SI $V_p \geq V_{bp}$: sucs := Ø (salir del ciclo).

[[AB = AB-{n}-sucs-aux]]

valor-a := V_p

```
VALOR-B(n,V_ap)
    sucs := SUCES(n)    [[AB := AB ∪ sucs; sucs-aux := sucs]]
    SI [sucs=Ø  v  Prof(n)=cota], ENTONCES,  valor-b := H(n)
    SI NO: V_p := Ø
           Hasta que sucs = Ø:
              ns := primer elemento de sucs
              sucs := sucs - {ns}
              val-a := VALOR-A(ns,V_p)
              Si  val-a < V_p : V_p := val-a.
                                      [[AB = AB-{ns}]]
              Si  V_p ≤ V_ap  : sucs := Ø (salir del ciclo).
                                      [[AB = AB-{n}-sucs-aux]]
    valor-b := V_p
```

2.2.1.- *Notas.*

Hacemos a continuación un breve comentario sobre el algoritmo anterior en sus dos versiones.

Se han omitido las acciones referentes al cálculo de sucesores, así como las referentes a la función de evaluación estática. Esto se debe a que ambas cuestiones son particulares de cada juego. En nuestro caso indicaremos estas acciones cuando se dé la aplicación al juego del tres en raya generalizado con deslizamiento. Como es de suponer se coincidirá con lo dicho cuando se aplicó el procedimiento Min-Max al citado juego.

Nótese que con respecto a la forma general, en esta forma detallada se omite la evaluación y poda del árbol. Esto es debido a que dichas acciones están recogidas de forma implícita en la recursividad que manifiesta el cálculo de los valores-α (valor-a) de los nodos MAX y el de los valores-β (valor-b) de los nodos MIN. En este último algoritmo, valor-a (respectivamente valor-b) es la variable que recoge los valores de la función VALOR-A(n,V_{bp}) (respectivamente VALOR-B(n,V_{ap})). En adelante usaremos indistintamente valor-a ó VALOR-A para designar los valores-α de los nodos MAX, y valor-b ó VALOR-B para los valores-β de nodos MIN.

Los pasos 1,2,3 y 4 del algoritmo (coinciden en ambas versiones, salvo en la notación empleada), constituyen el bloque central del algoritmo. Las diferencias entre una versión y otra están en las especificaciones dadas para el cálculo de lo que hemos denominado valores-α y valores-β.

La forma de operar del bloque central del algoritmo es parecida a la forma con que operaba la denominada etapa 3 del procedimiento Min-Max. Aquí, en el algoritmo Alfa-Beta, se incluyen las asignaciones V_{ap} = ( ) y V_{bp} = ( ). Estas variables representan, V_{ap} el valor-α provisional del padre, y V_{bp} el valor-β provisional del padre. La variable V_{ap} se utiliza para calcular el valor-b de los nodos MIN, mientras que V_{bp} se utiliza para calcular el valor-a de los nodos MAX.

Las diferencias de este algoritmo con respecto al Min-Max, se dan en la generación y evaluación del árbol. Estas diferencias saltan a la vista al comparar la etapa 2 del procedimiento Min-Max frente a lo que se ha denominado en Alfa-Beta, VALOR-A y VALOR-B. El motivo de estas diferencias, como ya se ha dicho, reside en el hecho de que mientras en Min-Max los procesos de generación y evaluación son independientes, en el sentido de que se genera el árbol completo y se evalúa el árbol completo, en Alfa-Beta ni se desarrolla ni se evalúa el árbol completo, sino que se efectúa lo que hemos denominado α-cortes y β-cortes. Estos cortes significan que se dejan de generar y de evaluar ciertos nodos (y los caminos que de ellos partan). Esto no es más que expresar el hecho conocido de que mientras en Min-Max, los procesos de generación y evaluación de nodos van separados, en Alfa-Beta van unidos, precisamente para poder establecer los cortes en el árbol.

Una vez comparados los dos algoritmos, pasemos a comentar algo más el algoritmo Alfa-Beta. Otra nueva variable que nos aparece es V_p . El hecho de arrastrar un valor provisional (V_p) tanto para calcular el VALOR-A como el VALOR-B de un nodo es debido a la posibilidad comentada de efectuar cortes en el árbol. Así, el algoritmo contempla un V_{ap} ó un V_{bp} para cada nodo en evaluación, un V_p que al final recoge el VALOR-A ó el VALOR-B del nodo en cuestión. También se usan las variables val-a y val-b (según se trate de nodos MIN ó MAX) que van cambiando según se

vaya recorriendo los sucesores del nodo en evaluación y en consecuencia calculando los VALOR-B ó VALOR-A de dichos sucesores. Finalmente, según sean V_p y val-a ó val-b, se rechaza el sucesor ó se actualiza V_p , y según sean V_p y V_{ap} ó V_{bp} se rechaza el nodo en evaluación (se efectua un corte) ó bien V_p sigue como mejor valor provisional.

Sobre el resto de variables y funciones que figuran en el algoritmo tenemos:

- sucs, SUCES, H, son particulares de cada juego. En la aplicación que daremos del algoritmo Alfa-Beta al juego Tres en raya Generalizado con Deslizamiento se utilizarán la variable sucs y las funciones SUCES y H (función de evaluación estática) de la misma forma en que se utilizaban en la aplicación del algoritmo Min-Max al citado juego.

- NI es la variable que recoge el nodo inicial. En el algoritmo trabajaremos también con pnodos (<profundidad,nodo>) y con otra entidad formada por <pnodo,evaluación del pnodo>, donde por evaluación del pnodo entenderemos su VALOR-A o su VALOR-B según se trate de un nodo MAX ó de un nodo MIN. Por último la variable ns es una variable de trabajo, que va recogiendo los pnodos sucesores del nodo en evaluación.

- La variable AB, es una variable auxiliar que no es parte, en sí misma, del algoritmo, y que se usarán para ver cuáles son los nodos del árbol que no son susceptibles de corte. Esta variable establecerá las diferencias entre los nodos expandidos por Min-Max y Alfa-Beta. Esta variable aparece entre un doble corchete en el algoritmo 2.2.

2.3.-Programa.

2.3.1.- *Notas.*

Veamos ahora la codificación, en lenguaje LISP, de un programa que implementa el Algoritmo Alfa-Beta. En primer lugar, aplicaremos el algoritmo al ejemplo dado en el apartado 2.1.2. En este programa tomaremos como funciones SUCES (cálculo de sucesores de un nodo) y H (función de evaluación estática) las reflejadas en la figura 2.1.2.1. Damos esta aplicación para poder observar el

cálculo de los valores-α y valores-β, así como la generación de árbol realizada. En la primera observación de los valores-α y valores-β, notaremos que la jugada elegida en ambos casos (Min-Max y Alfa-Beta) es la misma. Con la segunda observación del árbol generado, observaremos los cortes establecidos por Alfa-Beta con respecto a la generación exhaustiva que proporciona Min-Max. Esta observación se hace posible examinando la variable AB, significada en el algoritmo encerrada en un doble corchete, y que ya ha sido comentada antes.

Más adelante, en el apartado 2.4, veremos la aplicación del algoritmo Alfa-Beta al juego del Tres en raya Generalizado con Deslizamiento descrito en el apartado 1.4.

2.3.2.- *Codificación LISP del programa.*

```
;Asignación de variables globales
(setq AB ())
;1.- Función de cálculo de sucesores
(de SUCES (n)  (cond ((= n 1) '(11 12 13))
                     ((= n 11) '(111 112))
                     ((= n 111) '(1111 1112))
                     ((= n 1111) '(11111 11112 11113))
                     ((= n 1112) '(11121 11122))
                     ((= n 112) '(1121 1122 1123))
                     ((= n 1121) '(11211 11212))
                     ((= n 1122) '(11221))
                     ((= n 1123) '(11231 11232 11233))
                     ((= n 12) '(121))
                     ((= n 121) '(1211 1212))
                     ((= n 1211) '(12111 12112))
                     ((= n 1212) '(12121 12122 12123))
                     ((= n 13) '(131 132 133))
                     ((= n 131) '(1311))
                     ((= n 1311) '(13111 13112))
                     ((= n 132) '(1321 1322))
                     ((= n 1321) '(13211))
                     ((= n 1322) '(13221))
                     ((= n 133) '(1331))
                     ((= n 1331) '(13311 13312))
                     (t ()) ) )
```

;2.- Función de evaluación estática

```
(de H (n)
  (cond ((= n 11111) 1)     ((= n 11112) 2)
        ((= n 11113) -1)    ((= n 11121) -2)
        ((= n 11122) 3)     ((= n 11211) 1)
        ((= n 11212) 2)     ((= n 11221) 3)
        ((= n 11231) 1)     ((= n 11232) -2)
        ((= n 11233) 0)     ((= n 12111) 1)
        ((= n 12112) 0)     ((= n 12121) -1)
        ((= n 12122) 1)     ((= n 12123) 2)
        ((= n 13111) -1)    ((= n 13112) 1)
        ((= n 13211) 0)     ((= n 13221) 2)
        ((= n 13311) 1)     ((= n 13312) -2) ))
```

;3.- Programación del Procedimiento Alfa-Beta

;3.1.- Evaluación de nodos MAX, α-cortes

```
(de VALOR-A (n vbp)
  (let ((sucs (SUCES n)))
    (setq ab (append sucs ab))
    (cond ((null sucs) (H n))
          (t (let ((vp ()) (sucs-aux sucs))
               (while sucs-aux
                    (setq ns (car sucs-aux)
                          sucs-aux (cdr sucs-aux))
                    (setq val-b (VALOR-B ns vp))
                    (if (MENOR-IGUAL val-b vp)
                      (setq ab (remove ns (DIF ab (SUCES ns))))
                      (progn (setq vp val-b)
                           (if (MAYOR-IGUAL vp vbp)
                             (setq ab (remove n (DIF ab sucs))
                                   sucs-aux ()) ))))
               vp)))))
```

;3.2.- Evaluación de nodos MIN, β-cortes

```
(de VALOR-B (n vap)
  (let ((sucs (SUCES n)))
     (setq ab (append sucs ab))
     (cond ((null sucs) (H n))
          (t (let ((vp ()) (sucs-aux sucs))
                (while sucs-aux
                     (setq ns (car sucs-aux)
                           sucs-aux (cdr sucs-aux))
                     (setq val-a (VALOR-A ns vp))
                     (if (MAYOR-IGUAL val-a vp)
                       (setq ab (remove ns (DIF ab (SUCES ns))))
                       (progn (setq vp val-a)
                       (if (MENOR-IGUAL vp vap)
                         (setq ab (remove n (DIF ab sucs))
                              sucs-aux ()) ))))
               vp )))))
```

;3.3.- Funciones auxiliares

```
(de DIF (x y)
  (cond ((null x) ())
        ((member (car x) y) (DIF (cdr x) y))
        (t (cons (car x) (DIF (cdr x) y)) )) )
(de MENOR-IGUAL (a b)
  (if (or (null b) (null a)) ()  (<= a b)))
(de MAYOR-IGUAL (a b)
  (if (or (null b) (null a)) ()  (>= a b)))
```

;Documentación

;Funciones	Variables Locales	Variables Globales
;SUCES	n	AB
;H	n	
;VALOR-A	n, vbp, vp, val-b, sucs, sucs-aux, ns	
;VALOR-B	n, vap, vp, val-a, sucs, sucs-aux, ns	
;MENOR-IGUAL	a, b	
;MAYOR-IGUAL	a, b	
;DIF	x, y	

2.3.3.- *Notas.*

La variable AB, como ya se ha comentado, sólo sirve para mostrar cuáles son los nodos no rechazados en los cortes efectuados por el procedimiento. En concreto, reflejará la lista de los nodos, dados en la figura 2.1.3.4. Así, después de ejecutar el programa,

AB = (1, 11, 111, 1111, 11111, 11112, 11113, 12, 121,
1211, 12111, 12112, 13, 131, 1311, 13111),

que representa 14 nodos frente, a los 43 del desarrollo total.

La función SUCES toma como argumento un nodo, y da como resultado la lista de los sucesores de ese nodo. Nótese que en la aplicación anterior se desarrolla el árbol en su totalidad y en consecuencia no se utiliza la profundidad. La alternancia se conserva llamando a VALOR-B desde VALOR-A y viceversa. De esta manera se utiliza en el programa anterior la entidad nodo en vez de la entidad pnodo (profundidad, nodo).

La función H nos da la función de evaluación estática expresada también en la figura 2.1.2.1.

La función VALOR-A nos da la evaluación de los nodos MAX y los posibles α-cortes. Esta función toma como argumento un nodo y un valor provisional de la evaluación del padre (que será un nodo MIN) y devuelve la evaluación del nodo que toma como argumento. Esta función implementa el paso 5 del algoritmo Alfa-Beta en su forma detallada. Esto quiere decir, que para aplicar el programa anterior hemos de cargar el programa y dar la orden:

(VALOR-A '1 ())

con lo que se calculará la evaluación del nodo inicial que hemos etiquetado con el número 1. El resultado será 0 que corresponde al máximo valor de las evaluaciones de sus sucesores y que corresponde al nodo etiquetado con 12. Este es el resultado esperado a la vista del árbol y que coincide con la elección efectuada por el procedimiento Min-Max.

De forma análoga, la función VALOR-B calcula la evaluación de los nodos MIN y realiza los posibles β-cortes. Esta función opera de la misma forma que la anterior pero aplicada a nodos MIN. Nótese que el cambio de tipo de nodo (MAX ó MIN), implica, como se

indicaba en el paso 6 del algoritmo Alfa-Beta en su forma detallada (de la que es reflejo esta función), que el criterio de evaluación del nodo cambia así como el criterio para efectuar cortes. La función devuelve la evaluación del nodo que toma como argumento. Esta función no se usará para iniciar la ejecución del programa anterior ya que, como se ha dicho, estamos interesados en encontrar estrategias ganadoras para el jugador que comienza el juego y que denotamos por MAX.

Las funciones MENOR-IGUAL y MAYOR-IGUAL extienden los predicados $\leq$ y $\geq$, definidos entre números, al caso de que uno ó los dos números que entran como argumentos en ambos predicados, sean la lista vacía. Esto es necesario ya que la variable que se usa para arrastrar el valor provisional de la evaluación, toma este valor para significar el caso de que no exista valor provisional, como ocurre al principio de la ejecución del programa.

Por último, la función DIF, implementa la diferencia de conjuntos.

2.3.4.- *Calificación de variables.*

Variable	Valor
n	nodo $\in$ {1, 11, ..., 13312}
ns	nodo $\in$ {1, 11, ..., 13312}
AB	lista de nodos
vap	valor entero ó ( )
vbp	valor entero ó ( )
vp	valor entero ó ( )
val-a	valor entero ó ( )
val-b	valor entero ó ( )
suc	lista de nodos
suc-aux	lista de nodos
a	valores enteros ó ( )
b	valores enteros ó ( )
x	lista de nodos
y	lista de nodos

2.4.- Aplicacion del procedimiento Alfa-Beta al juego del Tres en Raya Generalizado con Deslizamiento.

En este apartado daremos la sintaxis de un programa en Lisp, que implemente el procedimiento Alfa-Beta (las funciones VALOR-A y VALOR-B junto con las funciones auxiliares), al juego descrito en el apartado 1.4. La descripción del juego nos dará las funciones de cálculo de sucesores y la función de evaluación estática de dicho juego. Estas, como es lógico, serán las mismas que las que se usaron para la aplicación del procedimiento Min-Max también a dicho juego. Con respecto al programa de aplicación del procedimiento Alfa-Beta al árbol de la figura 2.1.2.1, habrá que adecuar algunas variables, así como introducir la profundidad para limitar el desarrollo del árbol, pero ésto lo comentaremos luego.

2.4.1.- *Programa.*

```
;Aplicación del procedimiento Alfa-Beta al juego del Tres en Raya

;Generalizado con deslizamiento.

;Asignación a variables globales

(setq NI '(() ()) )

(setq lineas '( (1 2 3) (3 4 5) (5 6 7) (1 7 8) (10 11 12)
                (12 13 14) (14 15 16) (9 10 16) ) )

(plist 'vecinos '(1 (2 8) 2 (1 3 11) 3 (2 4) 4 (3 5 13) 5 (4 6)
                  6 (5 7 15) 7 (6 8) 8 (7 9 1) 9 (8 10 16)
                  10 (9 11) 11 (10 12 2) 12 (11 13) 13 (12 14 4)
                  14 (13 15) 15 (14 16 6) 16 (9 15)))

;1.- Función de evaluación estática

(de  H (pnodo)
       (let  ((p (PROF pnodo)) (pos+ (car (NODO pnodo)))
                               (pos- (cadr (NODO pnodo))) )
             (*  (if (evenp p) 1 -1)
                     (apply '+ (mapcar 'HAUX lineas)) )) )
```

```
(de  HAUX (linea)
        (let  ((aa (any '(lambda (x) (member x linea)) pos+))
               (bb (any '(lambda (x) (member x linea)) pos-)))
             (cond  ((equal pos+ linea) 100)
                    ((equal pos- linea) -100)
                    ((and (null aa) (null bb)) 0)
                    ((null bb) 1)
                    ((null aa) -1)
                    (t 0) )) )
```

```
;2.- Funciones de cálculo de sucesores
```

```
(de  SUCES (pnodo)
     (let ((p (PROF pnodo)) (nod (NODO pnodo)))
          (let ((n+ (NUM+ nod)) (n- (NUM- nod)))
               (cond ((= n+ 0) '( (1 ((1) ())) (1 ((2) ()))
                                  (1 ((9) ())) (1 ((10) ())) ))
                     ((> n+ n-) (mapcar '(lambda (x)
                                  (list (+ 1 p) x)) (PONE '- nod) ))
                     ((< n+ 3)  (mapcar '(lambda (x)
                                  (list (+ 1 p) x)) (PONE '+ nod) ))
                     (t  (mapcar '(lambda (x) (list (+ 1 p) x))
                         (DESLIZA (if (evenp p) '+ '-) nod)))))))
```

```
(de DESLIZA (jug nod)
  (let ((ocupados (append (car nod)(cadr nod))))
       (if (equal jug '+)
          (apply 'append (mapcar '(lambda (x)
                             (mapcar '(lambda (y)
                                (list (METE y (remove x (car nod)))
                                      (cadr nod)))
                             (DIF (get 'vecinos x) ocupados)))
                          (car nod)))
          (apply 'append (mapcar '(lambda (x)
                             (mapcar '(lambda (y)
                                (list (car nod)
                                   (METE y (remove x (cadr nod)))))
                             (DIF (get 'vecinos x) ocupados)))
                         (cadr nod))))))
```

```
(de  PONE (jug nod)
    (let ((I (DIF '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
                  (append (car nod) (cadr nod))  )))
        (cond ((equal jug '+)
                     (mapcar '(lambda (x)
                     (list (METE x (car nod))(cadr nod)))  I))
              (t (mapcar '(lambda (x)
                 (list (car nod) (METE x (cadr nod))))  I))) ))


(de METE (a l)
  (cond
     ((null l) (list a))
     ((= (length l) 1)
          (if (< a (car l)) (cons a l) (list (car l) a)))
     (t  (let ((b (car l)) (c (cadr l)))
          (cond  ((< a b) (list a b c))
                 ((< a c) (list b a c))
                 (t (list b c a)) )))))
```

3.- Programación del procedimiento Alfa-Beta

```
(de JUG-AB (nod)
  (setq cota (+ 1 (NUM+ nod)))
  (cadr (cadr (MEJOR (mapcar '(lambda (x) (list (VALOR-B x ()) x))
                              (SUCES (list 0 nod) )) )) ))


(de VALOR-A (n vbp)
  (if (terminal '- (NODO n))  -100
      (let ((suc (SUCES n))) (cond
         ((or (null suc) (= (PROF n) cota))  (H n))
         (t (let ((vp ()) (suc-aux suc))
            (while suc-aux
                 (setq ns (car suc-aux) suc-aux (cdr suc-aux))
                 (setq val-b (VALOR-B ns vp))
                 (if (MENOR-IGUAL val-b vp)  ()
                   (progn (setq vp val-b)
                       (if (MAYOR-IGUAL vp vbp)
                         (setq suc-aux ()) ))))
            vp))))) 
```

```
(de VALOR-B (n vap)
  (if (TERMINAL '+ (NODO n)) 100
      (let ((suc (SUCES n))) (cond
         ((or (null suc) (= (PROF n) cota)) (H n))
         (t (let ((vp ()) (suc-aux suc))
              (while suc-aux
                   (setq ns (car suc-aux) suc-aux (cdr suc-aux))
                   (setq val-a (VALOR-A ns vp))
                   (if (MAYOR-IGUAL val-a vp)  ()
                     (progn (setq vp val-a)
                          (if (MENOR-IGUAL vp vap)
                            (setq suc-aux ()) ))))
              vp ))))) )
```

4.- Funciones auxiliares y funciones de acceso

```
(de  NODO (pnodo)
          (cadr pnodo) )


(de  PROF (pnodo)
          (car pnodo) )


(de  NUM+ (nod)
          (length (car nod)) )


(de  NUM- (nod)
          (length (cadr nod)) )


(de DIF (x y)
     (cond ((null x) ())
           ((member (car x) y) (dif (cdr x) y))
           (t (cons (car x) (dif (cdr x) y)) )) )

(de TERMINAL (signo n)
        (if (equal signo '+) (member (car n) lineas)
                             (member (cadr n) lineas)))

(de MEJOR (l)
     (cond  ((= 1 (length l)) (car l))
            ((<= (caar l) (caadr l)) (MEJOR (cdr l)))
            (t (MEJOR (cons (car l) (cddr l)))) ))
```

```
(de MENOR-IGUAL (a b)
        (if (or (null b) (null a)) ()  (<= a b)))

(de MAYOR-IGUAL (a b)
        (if (or (null b) (null a)) ()  (>= a b)))

;Documentación
;Funciones              Variables locales       Variables globales
;H                      pnodo, pos+, pos-       NI, lineas, vecinos.
;HAUX                   línea, aa, bb, pos+,
;                       pos-
;SUCES                  pnodo, p, nod, n+, n-
;PONE                   jug, nod, I
;DESLIZA                jug, nod, ocupados,
;                       vecinos
;METE                   a, l, b, c
;JUG-AB                 nod, cota
;VALOR-A                n, vbp, suc, vp, ns,
;                       suc-aux, val-b
;VALOR-B                n, vap, suc, vp, ns,
;                       suc-aux, val-a
;NODO                   pnodo
;PROF                   pnodo
;NUM+                   nod
;NUM-                   nod
;DIF                    x, y
;TERMINAL               signo, n
;MEJOR                  l
;MENOR-IGUAL            a, b
;MAYOR-IGUAL.           a, b
```

2.4.2.- *Notas.-*

2.4.2.1.- H, HAUX.

Ambas funciones son exactas a sus homólogas en el procedimiento Min-Max; en consecuencia nos remitimos a lo dicho en el apartado 1.5.3. Ambas funciones implementan la función de evaluación estática usada en el juego.

2.4.2.2.- SUCES, PONE, DESLIZA, METE.

Como en el caso anterior, estas funciones son exáctas a sus homólogas en el procedimiento Min-Max, salvo la función PONE que representa una ligera variación en cuanto a la sintaxis empleada, no así en cuanto a el objetivo a cubrir (realizar la primera parte del juego), que es el mismo. La función PONE sigue ofreciendo los mismos resultados y tomando los mismos argumentos. Para el resto de funciones nos remitimos al apartado 1.5.4.

2.4.2.3.- JUG-AB.

Cubre los mismos objetivos que su homóloga en la programación del procedimiento Min-Max (allí la llamabamos JUGADA). Sin embargo, manifiesta las diferencias que a continuación comentamos. En primer lugar, mientras que JUGADA asumía una cota de profundidad fija en el desarrollo del árbol, que se recogía en la variable "cota", la función JUG-AB es la que asigna un valor a dicha variable. Además esta asignación es variable, siendo más pequeña al comienzo del juego, donde se demuestra haber menos dificultad en el juego, y siendo más grande en la etapa intermedia del juego (con todas las fichas sobre el tablero), donde existe más dificultad.

En segundo lugar, en vez de ordenar por evaluación los nodos sucesores del nodo inicial y elegir el primero de ellos, como ocurría en la función JUGADA, la función JUB-AB no ordena ninguna lista sino que elige, entre los nodos de una lista, aquél que tiene mejor evaluación.

2.4.2.4.- VALOR-A, VALOR-B.

Estas funciones son las que expresan el procedimiento Alfa-Beta. La función VALOR-A, toma coma como argumentos un pnodo y un "valor provisional para la evaluación del padre" en las variables "n" y "vbp". Es aplicable a nodos MAX, lo cual queda garantizado por las llamadas a VALOR-B. La función devuelve la evaluación del "pnodo", la cuál se puede tomar como nuevo valor provisional, ó bien puede servir para efectuar un α-corte. La forma de operar es la siguiente. En primer lugar, se comprueba si el nodo es perdedor para MAX, en cuyo caso asigna un valor de evaluación pequeño (en el programa figura el valor -100). En caso contrario, calcula los sucesores del nodo. Si el nodo no tiene

sucesores ó hemos alcanzado la cota de profundidad, se aplica la función de evaluación estática. En caso contrario, asignamos un "valor provisional" para la evaluación de los sucesores, y procede a evaluar uno a uno dichos sucesores. La evaluación de un sucesor provoca una llamada a la función VALOR-B, para calcular su valor. Si este valor es menor ó igual (según la función del mismo nombre) que el "valor provisional" con que se ha calculado, se sigue la evaluación del siguiente sucesor. En caso contrario (no menor ó igual), el valor de la evaluación de dicho nodo se toma como nuevo "valor provisional" en la evaluación del resto de sucesores. Luego se comprueba si dicho valor de evaluación, que ya es nuevo "valor provisional", es mayor ó igual (también según la función de igual nombre) que el "valor provisional de la evaluación del padre". En caso afirmativo, se detiene la evaluación del resto de sucesores (con la posible generación de nodos que esto conlleva) y se devuelve ese último valor de evaluación obtenido. En otras palabras, se produce un α-corte. El significado de las variables que emplea esta función con respecto a los conceptos antes mencionados, es el siguiente:

n = representa un pnodo

vbp = valor provisional de la evaluación del padre

suc, suc-aux = lista de los sucesores de n

ns = representa a cada pnodo sucesor de n

vp = valor provisional para la evaluación de los sucesores de n. Al igual que vbp, toma valores enteros o ().

val-b = recoge la evaluación de cada sucesor de n.

La función VALOR-B, actúa de la misma forma que VALOR-A pero desde el punto de vista del jugador que hemos designado por MIN. Las diferencias están en el criterio para dar valores al valor provisional para la evaluación de los sucesores del nodo a evaluar, y en el criterio para producir los cortes. En ambos casos adoptamos el criterio contrario, es decir, donde utilizamos la función MAYOR-IGUAL, empleamos MENOR-IGUAL y viceversa. También varía el criterio de nodo perdedor, comprobado al comienzo de la función. Aparecen las variables vap, para recoger en este caso el valor provisional de la evaluación del padre y val-a, para recoger los valores de la evaluación de los sucesores del nodo a evaluar. El resto de variables utilizadas tienen el mismo significado que las del mismo nombre usadas en la función VALOR-A.

2.4.2.5.- *Funciones auxiliares y funciones de acceso.*

Bajo este título, englobamos el resto de funciones definidas en el programa. En primer lugar trataremos las funciones de acceso, que son, NODO, PROF, NUM+ y NUM-. Todas estas funciones han sido utilizadas ya en la programación del procedimiento Min-Max, y se mantienen iguales en este programa de aplicación del procedimiento Alfa-Beta. Por tanto, nos remitimos al apartado 1.3.4 para la función PROF, y al apartado para las otras tres funciones.

Las funciones que denominamos auxiliares son: MEJOR, MENOR-IGUAL, MAYOR-IGUAL, DIF y TERMINAL. De éstas, las funciones MENOR-IGUAL, MAYOR-IGUAL y DIF, son las mismas que las utilizadas en el programa de aplicación del procedimiento Alfa-Beta que hemos desarrollado en el programa 2.3.2. En consecuencia nos remitimos a lo ya dicho.

Recuérdese que un "nodo" es una lista de dos listas, nodo = (L_1, L_2), donde L_1 y L_2 son a su vez listas formadas por 0, 1, 2 ó 3 números, que representan las posiciones del tablero ocupadas por MAX y MIN respectivamente. Un "pnodo" es un nodo y su profundidad, pnodo = (prof,nodo). También existe otra entidad, que no se ha nominado, y que está constituida por la evaluación de un pnodo y el pnodo.

La función MEJOR, toma como argumento una lista compuesta por estas últimas entidades constituidas por la evaluacion de un pnodo y el pnodo, y devuelve el elemento de la lista que corresponde al mayor valor de evaluación.

Finalmente la función TERMINAL, toma como argumento una variable que se denomina signo y que toma los valores + ó -, y un pnodo. Los valores + y - los utilizamos para designar las piezas de los jugadores MAX y MIN respectivamente. También toma como argumento un nodo. Esta función nos dice si un nodo es perdedor para MAX (mueve MAX y $L_2 \in$ lineas) ó no, si signo es igual a +, ó bien, si un nodo es perdedor para MIN (mueve MIN y $L_1 \in$ lineas) ó no.

2.4.2.6.- *Calificación de variables.*

Variables	Valor
pnodo, n, ns	pnodo
nod	nodo
linea	<1,2,3>, ,<9,10,16>
pos+, pos-	posición MAX, posición MIN
aa, bb	<>, <a>, <a,b>, <a,b,c>
p	profundidad
n+, n-	número de + , - de un nodo
jug, signo	+ ó -
I	lista de 1 a 16 enteros
a, b, c	un número entero del 1 al 16
l	lista de 1,2, ó 3 números del 1 al 16
vap, vbp, vp	un número entero ó ( )
val-a, val-b	un numero entero
suc, suc-aux	lista de pnodos
ni	(( ) ( ))
lineas	((1 2 3) ... (9 10 16))
vecinos	(1 (2 8) ... 16 (9 15))
ocupados	lista de 1 a 6 enteros del 1 al 16

CAPITULO IV.- α-β DEDUCCION.

§.1.- PLANTEAMIENTO.

§.2.- JUEGO DEDUCTIVO.

§.3.- α-β DEDUCCION.

§.4.- EJEMPLOS.

§.1.- PLANTEAMIENTO.

1.1.- *Notas.-*

Consideremos una base de conocimientos, expuestos en forma de cláusulas de Horn, esto es, mediantes de fórmulas del tipo:

$$A_{i_1} \wedge \ldots \wedge A_{i_{ni}} \longrightarrow A_i$$

$i = 1, \ldots, N$. Las fórmulas A_{ij} se llaman *premisas* de la cláusula, y la fórmula A_i es la *conclusion* de la cláusula.

Ciñéndonos al caso proposicional, estas fórmulas A_{ij} , A_i son proposiciones elementales (sin conectivas).

Denotaremos por For el conjunto de las proposiciones elementales que aparecen en las cláusulas de la base de conocimientos.

El problema que deben resolver los sistemas expertos que manejan este tipo de bases de conocimientos es el problema típico de la deducción:

1.2.- *Problema de deducción.*

Dados $A \in For$, y $D \subseteq For$, decidir si

$$\{CL, D\} \models A$$

donde CL es el conjunto de las cláusulas que no contienen fórmulas de D como conclusión.

Desde el punto de vista lógico, se trata de decidir si para cualquier valoración en la que tanto las cláusulas de CL como las fórmulas de D sean válidas, la fórmula A es también válida.

Las fórmulas de D pueden considerarse como "datos" ó "hechos establecidos"; la fórmula A es la pretendida conclusión. Podemos decir, simplemente, que el problema de deducción consiste en decidir si "A se deduce de D", naturalmente mediante las reglas de CL.

Para el problema descrito, excluimos en CL las cláusulas que contengan fórmulas de D como conclusión, pues es claro que no intervendrán en el proceso deductivo, al tratarse de fórmulas

válidas por hipótesis. Desde el punto de vista lógico, la justificación de tal exclusión se halla en que, si F es una fórmula válida, entonces, para cualesquiera fórmulas $F_1, \dots, F_k$, la fórmula $F_1 \wedge \dots \wedge F_k \longrightarrow F$ es una tautología.

1.3.- *Relación entre el problema de deducción y los juegos.*

1.3.1.- *Nota.*

Como ya se ha visto en capítulos precedentes, a un problema de deducción como el descrito puede asociársele un grafo Y/O, de forma que la solución afirmativa al problema de deducción sea equivalente a la existencia de un cierto g-camino en el grafo. Precisaremos más esta idea.

1.3.2.- *Definición.*

Pasamos a definir la relación entre el problema de deducción y los juegos.

Dado un problema de deducción por CL, D y A, consideremos el grafo Y/O $\langle For, R_{Cl} \rangle$ donde For es el conjunto ya descrito, y la relación $R_{Cl} \subseteq For \times \mathcal{P}(For)$ está definida por:

$$\langle A_i, \{A_{i_1}, \dots, A_{i_{ni}}\} \rangle \in R_{Cl} \longleftrightarrow$$

$$(A_{i_1} \wedge \dots \wedge A_{i_{ni}} \longrightarrow A_i) \in CL$$

En estas condiciones, reuniendo resultados ya conocidos, se tiene:

P.1 $(CL, D) \models A \longleftrightarrow$

P.2 $\exists$ g-camino de A a D en $\langle For, R_{Cl} \rangle \longleftrightarrow$

P.3 $\exists$ g-camino de $\langle A, 1 \rangle$ a D'' en $\langle For'', R''_{Cl} \rangle =$

$= AR\text{-}AA\,(GRA\text{-}ARB\,(\langle For, R_{Cl} \rangle)) \longleftrightarrow$

P.4 $\exists$ g-camino de $\langle A, 1 \rangle$ a D'' en el subgrafo del anterior que tome $\langle A, 1 \rangle$ como nodo inicial: notaremos este subgrafo como $J'(A)$ $(= \langle E', S' \rangle)$, por analogía con los juegos.

1.3.3.- *Algoritmo AA-AAJ.*

Sea J(A) = <E,S> el árbol Y/O alternado definido por J(A) = AA-AAJ (J'(A)), donde AA-AAJ es el algoritmo siguiente:

```
AA-AAJ (<E',S'>)
        E := E'
        S := S'
        P := Ø
        G := Ø
        Ter := {<nodo,número> sin sucesores de  E'}
        CICLO3

CICLO3 ()
        SI  Ter = Ø  ENTONCES : FIN  <E,S>
        SI  Ter ≠ Ø  ENTONCES :
             n := <nodo,número> primer elemento de Ter
             SI  n ∈ D'' ENTONCES :
                  SI  profundidad (n)  es impar
                       ENTONCES :  G := G U {n}
                  SI  profundidad (n) es par
                       ENTONCES : n' := <nodo,número-nuevo>
                                  E := E U {n'}
                                  S := S U {<n, {n'}>}
                                  G := G U {n'}
             SI  n ∉ D'' ENTONCES :
                  SI  profundidad (n)  es par
                       ENTONCES : P := P U {n}
                  SI  profundidad (n)  es impar
                       ENTONCES : n' := <nodo,número-nuevo>
                                  E := E U {n'}
                                  S := S U {<n, {m'}>}
                                  P := P U {n'}
```

1.3.3.1.- *Notas.*

Ter se obtiene mediante un algoritmo auxiliar, que básicamente recorre el conjunto E' y determina los <nodo,número> que no figuran como primer componente de los k-arcos que hay en S'. Estos <nodo,número> serán los <nodo,numero> que no tienen sucesores (elementos Terminales del grafo).

Aunque en el algoritmo no se ha expresado implícitamente, la profundidad ha de ser considerada con cada <nodo,número>.

La finalidad de este algoritmo es ir examinando los nodos sin sucesores de J'(A) y crear los conjuntos G y P, modificando eventualmente el árbol, con el siguiente criterio:

- P contendrá los nodos sin sucesores que no estén en D''; estos nodos, además, deberán tener profundidad par, por lo que, en caso necesario, se añade un arco complementario.

- G contendrá los nodos sin sucesores que sí estén en D''; además, estos nodos deberán tener profundidad impar, por lo que, como antes, en caso necesario se añade un arco.

La utilidad de este algoritmo se verá en el siguiente apartado.

1.3.3.2.- *Programa.*

El correspondiente programa Lisp, es el siguiente:

```
(de AA-AAJ (datos nodos sucesores)
   (setq nod () term ())
   (setq  nj (AUX-111 nodos)  sj sucesores  G ()  P ()
        terminales (FNSS nodos sucesores) )
   (setq ter-2 terminales)
   (CICLO3 terminales) )

(de AUX-111 (l)
        (if (null l) nod
            (setq nod (cons (cadar l) nod))  (AUX-111 (cdr l)) ))

(de FNSS (vn vs)
 (cond ((null vn) term)
       (t (if (PERTENECE-1 (car vn) vs) (FNSS (cdr vn) vs)
           (setq term (cons (car vn) term)) (FNSS (cdr vn) vs)))))

(de PERTENECE-1 (a l)
   (cond ((null l) ())
         ((equal (cadr a) (caar l)) t)
         (t (PERTENECE-1 a (cdr l)) )))
```

```
(de CICLO3 (ter)
 (if (null ter) (list 'nodos nj 'sucesores sj 'gan G 'per P)
     (setq ind (INDICE-2 (caadar ter) nj 0))
     (cond ((PERTENECE-2 (caadar ter) datos)
              (if (oddp (caar ter)) (setq G (cons (cadar ter) G))
                  (setq nj (append nj (list
                                   (list (caadar ter) (+ 1 ind)))
                        ))
                  (setq sj (append sj (list (list (cadar ter)
                                   (list (list (caadar ter)
                                          (+ 1 ind)))))
                        ))
                  (setq G (cons (list (caadar ter)
                                      (+ 1 ind)) G) )))
           (t
            (if (evenp (caar ter)) (setq P (cons (cadar ter) P))
                (setq nj (append nj (list (list (caadar ter)
                           (+ 1 ind))) ))
                (setq sj (append sj (list (list (cadar ter)
                           (list (list (caadar ter) (+ 1 ind))))))
                           )
                (setq P (cons (list (caadar ter) (+ 1 ind)) P) )))
                       )
     (CICLO3 (setq ter (cdr ter)))))

(de PERTENECE-2 (a l)
   (cond ((null l) ())
         ((equal a (car l)) t)
         (t (PERTENECE-2 a (cdr l)) )))

(de INDICE-2 (m lista vi)
   (if (null lista) vi
       (let ((nodo (caar lista)) (numero (cadar lista)) )
        (if (and (equal m nodo) (< vi numero))
            (INDICE-2 m (cdr lista) numero)
            (INDICE-2 m (cdr lista) vi) ))))
```

1.3.3.3.- *Notas.*

La variable datos recoge los nodos hipótesis. En el ejemplo II-1.1 datos = (BS BP). Las variables nodos y sucesores recogen la salida de AR-AA modificado. Esta modificación se concreta en considerar la profundidad junto con los <nodo,número>. En el ejemplo citado tenemos,

nodos = ((0 (PD 1))...(5 (BP 2)))

sucesores = (((PD 1) ((PE 1)))...((MC 2) ((BS 2) (BP 1)))...
((BA 1) ((BP 2))))

Las variables nj , sj representan los nuevos conjuntos de nodos y sucesores; G y P son los conjuntos creados por el algoritmo.

La función AUX-111 elimina la profundidad en los elementos de la lista nodos.

La función FNSS devuelve la lista de pnodos (prof nodo) que no tienen sucesores. Esta lista es el valor de la variable terminales, es decir, terminales = ((2 (CU 1)) (2 (BS 1)) (4 (BS 2)) (4 (BP 1)) (5 (BP 2))). La variable ter-2 también recoge la lista de pnodos sin sucesores. La función PERTENECE-1 devuelve t si un pnodo tiene sucesores, () en caso contrario.

La función CICLO3 realiza la corrección de la profundidad de los nodos terminales, añadiendo si es preciso nuevos nodos y nuevos arcos. También determina los conjuntos G y P.

La función PERTENECE-2 es parecida a PERTENECE-1. PERTENECE-2 es como la función "member" de LISP, pero dando True o False, mientras que PERTENECE-1 determina la pertenencia de un nodo a una lista, teniendo en cuenta que sus argumentos vienen dados como

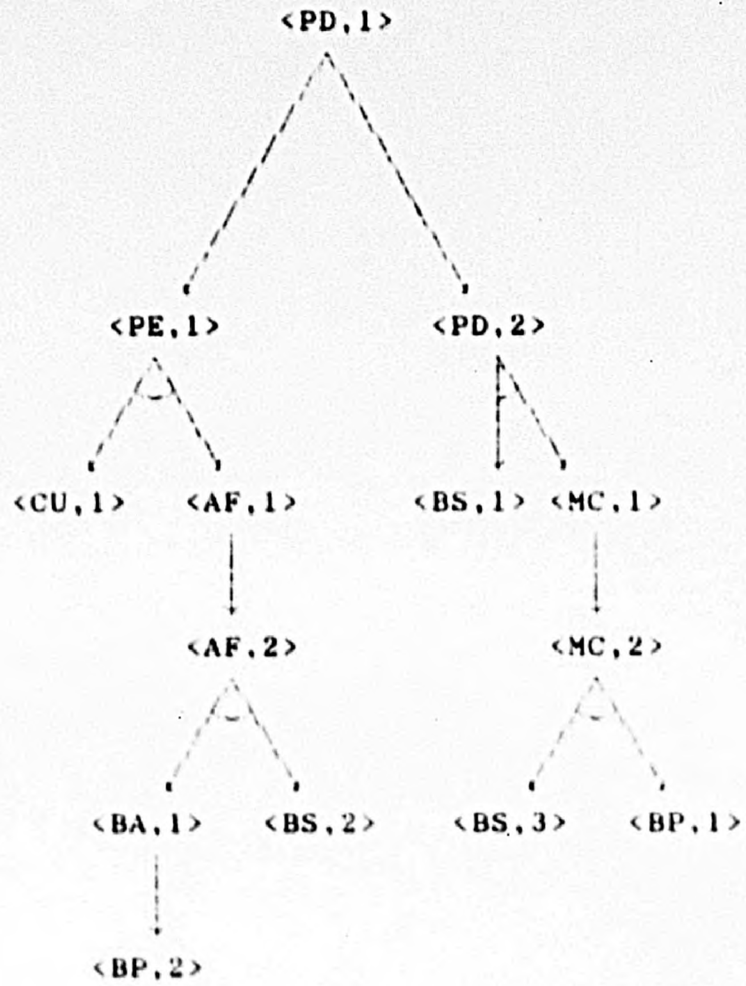
(profundidad (nodo numero)), y

((nodo numero) ((nodo numero)...))

1.4.- *Ejemplo.*

La aplicación de AR-AA y GRA-ARB para el ejemplo II-1.1 nos da el árbol Y/O alternado:

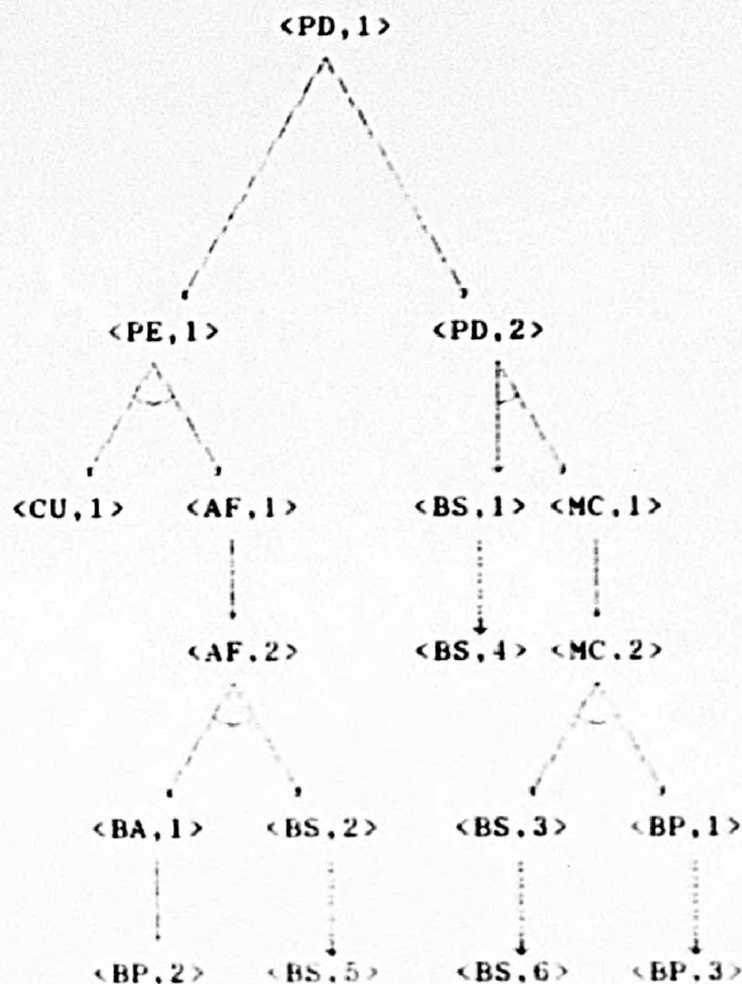
Figura 1.4.1.



con $D'' = \{\langle BS, 1\rangle, \langle BS, 2\rangle, \langle BS, 3\rangle, \langle BP, 1\rangle, \langle BP, 2\rangle\}$.

La aplicación de AA-AAJ a árbol anterior nos devuelve

Figura 1.4.2.



pues:

- el único nodo terminal no de D'' es <CU, 1>; como tiene profundidad par, se incluye en P sin ninguna modificación; así: P = {<CU, 1>}.

- Los demás nodos terminales están en D''; el único que tiene profundidad impar es <BP, 2>; para los demás es preciso añadir un arco suplementario, quedando finalmente G = {<BS, 4>, <BS, 5>, <BS, 6>, <BP, 2>, <BP, 3>}.

1.4.1.- *Notas.*

Es evidente que P.4 de la definición 1.3.2. y por tanto el problema de deducción P.1, es equivalente a la existencia en J(A) de g-camino de <A, 1> a G.

§.2.- JUEGO DEDUCTIVO.

Acabamos de ver cómo a partir de un problema de deducción $(CL, D) \models A$ hemos llegado al problema equivalente de existencia de g-camino en un cierto árbol Y/O $J(A) = \langle E, S\rangle$. Además en el proceso hemos construido unos conjuntos G y P.

Nuestra idea es asociar un juego al proceso deductivo; en este sentido, los estados ganadores, G, corresponderán a fórmulas que sabemos que son válidas, y los perdedores, P, a fórmulas que resultan indemostrables en nuestra base de conocimientos.

2.1.- *Definición.*

Dado un problema de deducción como el descrito, esto es, dados $J(A) = \langle E, S\rangle$, G, y P, se llama *juego deductivo asociado* al problema de deducción al juego

$$J_A = \langle E,R,T,G,P\rangle$$

donde T = {nodos de profundidad par}, y R viene dada por:

$$\forall s \in E \quad R(s) = \bigcup S(s).$$

2.2.- *Notas.*

Es trivial comprobar que J_A cumple las condiciones de la definición I-2.1.

El juego J_A tal como está construido, no es interpretable en grafos. Sin embargo, veremos que es isomorfo a un juego, que resulta ser una copia de otro, que si es interpretable en grafos, en la que se fija el jugador que comienza, o lo que es lo mismo, se obliga a que el nodo inicial sea de T. (Ver Nota I.2.26).

En efecto, dado $J_A = \langle E,R,T,G,P\rangle$, consideremos el juego $\langle E_1,R_1,T_1,G_1,P_1\rangle$ dado por:

$E_1 = E \times \{0,1\}$

R_1 dada por: $\langle n_1,0\rangle \, R_1 \, \langle n_2,1\rangle \iff n_1 \, R \, n_2$, y

$\langle n_1,1\rangle \, R_1 \, \langle n_2,0\rangle \iff n_1 \, R \, n_2$

$T_1 = \{\langle n,1\rangle \in E_1\}$

$G_1 = \{\langle n,0\rangle \in E_1 : R(n) = \emptyset\}$

$P_1 = \{\langle n,1\rangle \in E_1 : R(n) = \emptyset\}$.

Es claro que este juego es interpretable en grafos, sin más que tomar $N = E$, $R' = R$, y $F = \{n \in E : R(n) = \emptyset\}$.

Consideremos ahora la copia $\langle E_2, R_2, T_2, G_2, P_2\rangle$ dada por:

$E_2 = \{\langle n,0\rangle \in E_1 : prof(n) \text{ impar}\} \cup$

$\{\langle n,1\rangle \in E_1 : prof(n) \text{ par}\}$

$R_2 = R_1|_{E_2}$

$T_2 = T_1 \cap E_2$

$G_2 = G_1 \cap E_2$

$P_2 = P_1 \cap E_2$

Es claro también que la aplicación $\Phi : E \longrightarrow E_2$ definida por:

$$\Phi(n) = \begin{cases} \langle n,0\rangle & \text{si } prof(n) \text{ es impar} \\ \langle n,1\rangle & \text{si } prof(n) \text{ es par} \end{cases}$$

es un isomorfismo de juegos. (Ver I.2.29)

2.3.- *Teorema.*

En las condiciones anteriores:

$(CL, D) \models A \leftrightarrow \exists$ estrategia ganadora para $\langle A, 1\rangle$ en el juego J_A

Prueba:

Basta demostrar que

$\exists$ estrategia ganadora para $\langle A, 1\rangle$ en el juego J_A $\leftrightarrow$

$\exists$ g-camino de $\langle A, 1\rangle$ a G en $J(A) = \langle E, S\rangle$

y para probar esto, basta ver que el grafo $\langle E, S\rangle$ que se considera en el teorema II-2.10 es precisamente $J(A)$, y aplicar el propio teorema II-2.10.

Para evitar confusiones, llamemos S^* la relación notada como S en II-2.10.

Si $s \in T$:

s tiene profundidad par en $J(A)$ $\Rightarrow$ s es origen únicamente de 1-arcos (por ser $J(A)$ alternado) $\Rightarrow$

$S(s) = \{\{n_1\}, \ldots, \{n_k\}\} \Rightarrow$

$R(s) = \bigcup S(s) = \{n_1, \ldots, n_k\} \Rightarrow$

$S^*(s) = \{\{n\} : n \in R(s)\} = \{\{n_1\}, \ldots, \{n_k\}\} = S(s)$.

Si $s \notin T$:

s tiene profundidad impar en $J(A)$ $\Rightarrow$

s es origen únicamente de un k-arco $\Rightarrow$

$S(s) = \{\{n_1, \ldots, n_k\}\}$ $\Rightarrow$

$R(s) = \bigcup S(s) = \{n_1, \ldots, n_k\}$ $\Rightarrow$

$S^*(s) = \{R(s)\} = \{\{n_1, \ldots, n_k\}\} = S(s)$

con lo que $\langle E, S^* \rangle = \langle E, S \rangle$. ■

2.4.- *Notas.*

Hemos llegado así al punto crucial de este trabajo: dado un problema de deducción $(CL, D) \models A$, hemos construido un juego J_A y un estado del juego $\langle A, 1\rangle$, de manera que el problema de deducción se resuelve afirmativamente si y sólo si existe una estrategia ganadora del juego para ese estado.

Ambos problemas presentan aspectos y dificultades similares. Si pudiésemos considerar el árbol completo, "viendo" la estructura del problema podríamos sugerir estrategias de resolución. Pero forzosamente, la máquina limita esta posibilidad y, sobre todo, la máquina no puede "ver" ni extraer conclusiones de manera intuitiva (no formalizada), sino que examinará todas las posibilidades, incluso las que desecharíamos de un simple vistazo.

A lo largo de la presente memoria hemos insistido en la dificultad de encontrar estrategias ganadoras para juegos. Hablaremos aquí de la dificultad del problema de deducción.

En primer lugar repetiremos que nos hemos ceñido al caso proposicional. Hay una diferencia fundamental con el caso de la lógica de primer orden: la lógica proposicional es decidible y la de primer orden es indecidible. Esta diferencia esencial resulta, en las máquinas, de tipo secundario: no importa el carácter decidible o no; lo que nos importa es que, si el programa no queda bloqueado, nos dará una respuesta positiva o negativa. Esta es, por otra parte, una perogrullada: si demostramos un resultado (con ordenador o con papel y lápiz), el resultado queda probado por más que la lógica en cuestión sea indecidible, y si no logramos demostrar un resultado, la prueba quedará en el aire, por más que la lógica sea decidible. En otras palabras: lo que nos interesa es conocer la demostración del resultado, y la justificación de cada

paso de la demostración; en las máquinas, los sistemas expertos deben poder justificar sus diagnósticos o conclusiones.

Desde el punto de vista algorítmico, la lógica de primer orden precisa, para la aplicación de las reglas, del proceso de unificación de variables y términos, unido al de "skolemización" de las fórmulas, para después sufrir un tratamiento similar al proposicional. Así, la parte deductiva es similar en ambos casos.

Llegados a este punto, diremos que, hasta donde conocemos, hay dos maneras básicas de enfocar mecánicamente el problema de deducción, cada una con una cantidad de variantes y algoritmos relacionados: Resolución y AO (ver por ejemplo, [NI],[DE])

La Resolución se basa en el principio de Resolución (Robinson, 1965); se añade la negación de la conclusión al conjunto de cláusulas, y se intenta demostrar la inconsistencia del conjunto así obtenido. Para este fin, se van añadiendo al conjunto las "resolventes" de cada dos cláusulas (la resolvente de dos cláusulas es la cláusula obtenida uniendo las dos y suprimiendo las fórmulas elementales que aparezcan negadas y afirmadas), y proseguir así hasta que aparezca la cláusula vacía, lo que indicará la inconsistencia y por tanto la respuesta afirmativa al problema de deducción planteado.

Las variantes del método se deben, fundamentalmente, al orden en que se emparejan las cláusulas para ir obteniendo las sucesivas resolventes.

Se trata de un método bastante bueno, y muy extendido: por ejemplo, el lenguaje PROLOG no es más que una realización concreta de la resolución.

Tiene una dependencia muy grande del orden elegido: PROLOG, por ejemplo, utiliza en su estrategia el orden en que hayamos escrito las cláusulas en el programa: esto hace que a veces un programa quede bloqueado por una pregunta que, con otro orden de escritura, tendría una respuesta trivial.

AO es un algoritmo de búsqueda en grafos Y/O. (AO son las iniciales de AND/OR), que proporciona un g-camino desde un nodo dado a un conjunto de nodos objetivos. AO proporciona g-caminos de coste minimal, suponiendo que los nodos y los conectores tienen asociado un coste.

AO va creando un grafo de búsqueda que contiene los nodos ya desarrollados y los arcos correspondientes, junto con un grafo solución provisional que contiene las mejores soluciones parciales halladas hasta el momento.

AO presenta variantes basadas en ciertas relaciones de acotación de las funciones de evaluación de nodos y arcos, así como en el orden de expansión de nodos. Se trata de un algoritmo costoso , de relativamente poco uso para problemas de deducción; se usa algo más para juegos, pero, debido a su naturaleza exhaustiva, suele reservarse para finales de partidas.

Hemos visto así las dificultades tanto de encontrar estrategias ganadoras para juegos, como de resolver el problema de deducción.

En el capítulo anterior hemos visto cómo la estrategia α-β nos permitía hacer un movimiento de un juego que, sin formar parte con seguridad de una estrategia ganadora, es un buen movimiento en función de la profundidad del examen y de las funciones heurísticas empleadas.

A la vista del teorema 2.3 cabe preguntarse sobre el sentido que puede tener la aplicación de α-β al juego J_A para tratar el problema de deducción. Dedicaremos a esto el siguiente apartado.

§.3.- α-β DEDUCCION.

3.1.- Notas.-

Llamando α-β *Deducción* al proceso resultante de la aplicación de α-β a los procesos deductivos, lo primero que se observa es que la α-β deducción nos da un paso por etapa, esto es, la respuesta típica del sistema es del tipo:

"para demostrar tal fórmula es conveniente demostrar tal fórmula (o tales fórmulas)".

La segunda característica importante viene dada por las funciones heurísticas de evaluación de nodos. Al ser los nodos terminales bien de F o bien de G, es claro que solo necesitamos dos valores, por ejemplo -1 y 1. Los nodos de F provienen de las proposiciones indemostrables para la base de conocimientos (que si no podemos evitarlas nos hacen perder el juego, esto es, no nos

permiten realizar la demostración), y los nodos de G provienen de las hipótesis (que si nos reducimos a ellos nos hacen ganar el juego, esto es, terminar la demostración con éxito).

Naturalmente, esto no impide que puedan asignarse valores estáticos distintos: en el ejemplo II-1.1 de los préstamos, es perfectamente posible que sea más fácil para un banco el evaluar si un cliente tiene buen sueldo que el evaluar si el cliente es buena persona. Así, una demostración basada en que el cliente tenga buen sueldo será "más barata" que otra que precise demostrar que se trata de una buena persona.

Por último, señalaremos que α-β no garantiza que el movimiento pertenezca a una estrategia ganadora, pero nos aconseja un buen movimiento después de desechar una buena cantidad de movimientos inútiles. Del mismo modo, la α-β deducción producirá un resultado de semidecisión: si demostramos la fórmula, desde luego que la fórmula es demostrable, pero si no logramos la demostración, no podemos asegurar que la fórmula es indemostrable. La ventaja estriba en que, cuando se realice la demostración, se habrá realizado con un coste bastante menor en tiempo y memoria que mediante un proceso exhaustivo.

3.2.- *Programas AEDED y ABDAUX.*

Nos planteamos ahora la reunión de los distintos resultados obtenidos en la memoria, para construir un programa que implemente el proceso del que hemos estado hablando en este apartado, y que hemos denominado α-β deducción.

3.2.1.- *Programa ABDED.*

```
(de abded (e s datos objetivo)
   (load "gra-arb.ll")
   (gra-arb e s)
   (load "ar-aa-2.ll")
   (ar-aa ee ss)
   (load "aa-aaj.ll")
   (aa-aaj datos eee sss)
   (setq lista_objetivos (list (list objetivo l)))
   (load "abdaux.ll")
   (ciclo4) )
```

3.2.2.- *Notas.*

Este programa toma como argumentos un grafo Y/O, <e s>, un conjunto datos y un objetivo, que representan las reglas, las premisas y la conclusión correspondientes al problema de deducción. El programa realiza los pasos previos a la aplicación del algoritmo α-β, es decir, en primer lugar transforma el grafo Y/O en un árbol Y/O y ocasionalmente, modifica el conjunto datos. Para ello carga y ejecuta el programa GRA-ARB. La salida de este programa la notamos por ee y ss, y será a su vez la entrada del siguiente paso. Respecto al programa GRA-ARB, nos remitimos al capítulo II, y más concretamente al apartado II-3.6.

En segundo lugar, se transforma el árbol Y/O, obtenido en el paso anterior, en un árbol Y/O alternado y ocasionalmente, también modifica el conjunto datos, probablemente modificado en el paso anterior. En este caso se carga el programa AR-AA-2 y lo ejecuta. La salida de este programa la notamos por eee y sss, que será a su vez la entrada en el siguiente paso. Respecto a este nuevo programa, nos remitimos al capítulo II en su apartado II-3.7. Solamente hacemos la salvedad, ya comentada en el presente capítulo, de que hemos modificado el programa original que allí figura, incluyendo en los elementos del conjunto de nodos eee la profundidad de dichos nodos. Este cambio es debido a que en el siguiente paso nos es necesaria la profundidad de los nodos. Dicha modificación del programa es mínima respecto a la codificación dada en II-3.7, y por tanto, no incluimos aquí la codificación correspondiente.

El último paso previo a la aplicación de α-β, lo da el programa AA-AAJ, que consiste en adecuar la profundidad de los nodos terminales y la formación definitiva de los conjuntos G y P utilizados en la definición del juego deductivo. Este programa toma como argumento la salida de AR-AA-2 y añade, si procede, copias de nodos terminales, formando definitivamente los conjuntos G y P. Para más explicaciones nos remitimos al apartado 1.3.3.3 de este capítulo.

Por último, el programa AB-DED carga el programa AB-DAUX que aplica el algoritmo α-β a la salida AA-AAJ, para realizar la demostración, si es posible, del problema de deducción ó bien dar el mensaje de "no puedo hacer la demostración".

Antes de dar la codificación del programa AB-DED, puntualizamos un aspecto que tratamos cuando hablabamos de árbol de la jugada hasta una profundidad igual a cota en el apartado III-1.1.1. En aquella ocasión, considerábamos una cota de profundidad en el desarrollo del árbol del juego. Ahora vamos a obviar la cota de profundidad pues pensamos que es mejor que las limitaciones en el desarrollo del árbol de juego las ponga la propia máquina, en vez de ponerlas nosotros explícitamente en el programa. De esta manera la α-β deducción va a ofrecer tres posibles resultados. La primera posibilidad es que se logre la demostración del problema de deducción. Para el caso de que esta demostración no se alcance tenemos a su vez dos posibilidades, una es que el programa nos dé el mensaje "no puedo hacer la demostración" como ya hemos dicho antes, y otra consiste en que el programa quede bloqueado por falta de memoria para resolver el problema de deducción en cuestión. En este último caso no podemos decir si el problema de deducción tiene o no demostración.

Este último programa, que comentaremos más ampliamente en el siguiente apartado, comienza creando una lista_objetivos que se obtiene del objetivo original, incluyendo en el nodo el número 1. La finalidad es que exista compatibilidad entre las entidades tratadas por los distintos programas (nótese que tanto AR-AA-2 como AA-AAJ tratan con <nodo,número> en vez de simplemente nodo). El programa se ejecuta llamando a la función CICLO4, que sera también tratada en el siguiente apartado.

3.2.3.- *Programa AB-DAUX.*

```
(de CICLO4 ()
 (if (null lista_objetivos)
      (print "Fin de la demostracion")
      (setq obj (car lista_objetivos)
            lista_objetivos (cdr lista_objetivos)
            objl (JUGADA obj)
            sucs (SUCES3 objl) )
      (print "para demostrar " obj " hay que demostrar " objl)
      (if (and (null sucs) (member objl g))
         (print objl " es un dato")
         (print "para demostrar " objl " hay que demostrar" sucs))
      (if (every 'TEST sucs) (CICLO4))))
```

```
(de TEST (n)
   (cond ((not (member n p)) (setq lista_objetivos
                                   (cons n lista_objetivos)))
         (t (progn  (print "no puedo hacer la demostracion")
              () ) ) ) )

(de JUGADA (n)
   (cadr (MEJOR (mapcar '(lambda (x) (list (VALOR-B x ())) x))
                             (SUCES3 n))))))

(de MEJOR (l)
   (cond  ((= 1 (length l)) (car l))
          ((<= (caar l) (caadr l)) (MEJOR (cdr l)))
          (t (mejor (cons (car l) (cddr l)))))))

(de DIF (x y)
    (cond ((null x) ())
       ((member (car x) y) (DIF (cdr x) y))
       (t (cons (car x) (DIF (cdr x) y)) )) )

(de VALOR-A (n vbp)
         (let ((suc (SUCES3 n)))
          (cond ((null suc) (h n))
                (t (let ((vp ()) (suc-aux suc))
           (while suc-aux
               (setq ns (car suc-aux) suc-aux (cdr suc-aux))
               (setq val-b (VALOR-B ns vp))
               (if (MENOR-IGUAL val-b vp)  ()
                   (progn (setq vp val-b)
                          (if (MAYOR-IGUAL vp vbp) (setq suc-aux ()))
           ))))   vp)))))

(de VALOR-B (n vap)
          (let ((suc (SUCES3 n)))
            (cond ((null suc) (H n))
                  (t (let ((vp ()) (suc-aux suc))
             (while suc-aux
                (setq ns (car suc-aux) suc-aux (cdr suc-aux))
                (setq val-a (VALOR-A ns vp))
                (if (MAYOR-IGUAL val-a vp)  ()
                    (progn (setq vp val-a)
                           (if (MENOR-IGUAL vp vap) (setq suc-aux ()))
             ))))    vp )))))
```

```
(de MENOR-IGUAL (a b)
        (if (or (null b) (null a)) () (<= a b)))

(de MAYOR-IGUAL (a b)
  (if (or (null b) (null a)) () (>= a b)))

(de H (n)
    (cond ((member n g) 1)
          ((member n p) -1)
          (t 0)))

(de SUCES3 (n)
   (prog (result)
        (mapcar '(lambda (x)
                   (if (equal n (car x))
                        (setq result (append (cadr x) result)))) sj)
        (return result) ) )
```

3.2.4.- *Notas.*

La función CICLO4, actúa de la siguiente forma: cuando lista_objetivos sea vacía, devuelve el mensaje "Fin de la demostración", para indicar que se ha demostrado el problema de deducción dado. En caso contrario, asigna a la variable obj el primer elemento de lista_objetivos, elimina el primer elemento de lista_objetivos, asigna a obj1 el resultado de aplicar el algoritmo α-β a obj y a sucs los sucesores de obj1. Con esto pasamos a demostrar obj1 en vez de obj. Ahora bien, si sucs es vacío ó obj1 está en G, no tendremos que demostrar nada pues obj1 es un dato de nuestro problema. En caso contrario, demostraremos sucs en vez de obj1. Antes de demostrar sucs, lo cual haremos repitiendo de nuevo CICLO4 aplicado ahora a distintos valores de lista_objetivos, hemos de ver si no hemos llegado a algún nodo de P, en cuyo caso no podremos hacer la demostración y por consiguiente se debería terminar la ejecución del programa. Para este fin, aplicamos la función TEST a cada elemento de sucs.

La función TEST evalúa si un nodo terminal no esta en P, en cuyo caso lo añade a lista_objetivos. En caso contrario termina el proceso de demostración, indicando que tal demostración no es posible, es decir, indica que el problema de deducción considerado no tiene demostración.

Las funciones JUGADA, VALOR-A y VALOR-B, implementan el algoritmo α-β, y ya fueron comentadas en el apartado III-3.3.3. Lo mismo ocurre con las funciones auxiliares que se utilizan el la programación del algoritmo α-β. Estas funciones recordemos que eran, MEJOR, MAYOR-IGUAL, MENOR-IGUAL y DIF.

La función SUCES3 utiliza la lista sj tal como la da AA-AAJ. Esta función toma como argumento un <nodo,número> y devuelve la lista de <nodo,número> sucesores del anterior. Esta función será particular de cada problema de deducción que se considere dependiendo de la configuración de la lista sj que se tenga. Esta lista sj se obtiene de la lista s que toma como argumento AB-DED, por las sucesivas transformaciones a que es sometida por GRA-ARB, AR-AA-2 y AA-AAJ.

Finalmente, la función H es la función de evaluación estática usada en la aplicación del algoritmo α-β a la deducción. Esta función, como ya se ha dicho en las notas 3.1, da valores de evaluación a los nodos terminales de acuerdo al criterio: si un nodo está en P su valor es -1, mientras que si el nodo está en G se da valor 1. La función H utiliza los conjuntos G y P tal y como los da AA-AAJ.

3.3.- *Nota final.*

Esta memoria es un trabajo académico. Se ha realizado siguiendo criterios y esquemas propios de la ciencia "pura". Así, hemos presentado la formalización de muchos conceptos que se presentan normalmente de manera ingenua, se ha producido un proceso de abstracción que nos ha llevado a presentar como idénticas dos estructuras que aparecían como distintas, y hemos realizado la aplicación de una estrategia de juegos a la resolución de problemas deductivos.

Nótese que, aunque para nosotros el problema lógico representa un objetivo más importante que el lúdico, podría pensarse en una aplicación de las estrategias de razonamiento automático a la búsqueda de estrategias ganadoras para juegos.

Aclararemos igualmente que la memoria no pretende ser ni presentar un producto comercial. Así, los algoritmos y programas que se presentan van siempre acompañando conceptos e ilustrando dificultades teóricas; algunos pueden resultar innecesarios desde

un punto de vista operativo. Tampoco hemos realizado un estudio de la complejidad de los algoritmos presentados con objeto de mejorarla: se ha preferido que los algoritmos, más que rápidos, sean claros e inteligibles.

Presentaremos, para terminar, algunos ejemplos de α-β deducción.

§.4.- EJEMPLOS α-β DEDUCCION.

4.1.- *Ejemplo*

Nuestro primer ejemplo será el que hemos arrastrado a lo largo de todo el trabajo:

Cláusulas:

BS ∧ BA → AF
AF ∧ CU → PE
PE → PD
BP ∧ BS → MC
BS ∧ MC → PD
BP → BA

Datos: {BS,BP}

Pretendida conclusión: PD

Con los adecuados e. s. la llamada

(abded e s '(bs bp) 'pd)

produce la salida:

```
para demostrar (pd 1) hay que demostrar (pd 2)
para demostrar (pd 2) hay que demostrar ((bs 1) (mc 1))
para demostrar (mc 1) hay que demostrar (mc 2)
para demostrar (mc 2) hay que demostrar ((bs 2) (bp 1))
para demostrar (bp 1) hay que demostrar (bp 3)
(bp 3) es un dato
para demostrar (bs 2) hay que demostrar (bs 5)
(bs 5) es un dato
para demostrar (bs 1) hay que demostrar (bs 6)
(bs 6) es un dato
Fin de la demostracion
```

4.2.- *Ejemplo.*

Cáusulas:

```
B ∧ D ∧ E  →  F
    X ∧ A  →  H
    G ∧ D  →  A
        C  →  D
    C ∧ F  →  A
    X ∧ C  →  A
        B  →  X
    X ∧ B  →  D
        D  →  E
```

Datos: (B,C)

Pretendida conclusion: H

Salida del programa:

```
para demostrar (h 1) hay que demostrar (h 2)
para demostrar (h 2) hay que demostrar ((x 1) (a 1))
para demostrar (a 1) hay que demostrar (a 3)
para demostrar (a 3) hay que demostrar ((c 1) (f 1))
para demostrar (f 1) hay que demostrar (f 2)
para demostrar (f 2) hay que demostrar ((b 3) (d 2) (e 1))
para demostrar (e 1) hay que demostrar (d 3)
para demostrar (d 3) hay que demostrar ((c 5))
para demostrar (c 5) hay que demostrar (c 6)
(c 6) es un dato
para demostrar (d 2) hay que demostrar (c 4)
(c 4) es un dato
para demostrar (b 3) hay que demostrar (b 12)
(b 12) es un dato
para demostrar (c 1) hay que demostrar (c 8)
(c 8) es un dato
para demostrar (x 1) hay que demostrar (b 1)
(b 1) es un dato
Fin de la demostracion
```

4.3.- *Ejemplo.*

Cláusulas:

Las mismas que en 4.2.

Datos: (B)

Pretendida conclusión: F

Salida del programa:

```
para demostrar (f 1) hay que demostrar (f 2)
para demostrar (f 2) hay que demostrar ((b 1) (d 1) (e 1))
para demostrar (e 1) hay que demostrar (d 4)
para demostrar (d 4) hay que demostrar ((x 2) (b 4))
para demostrar (b 4) hay que demostrar (b 6)
(b 6) es un dato
para demostrar (x 2) hay que demostrar (b 5)
(b 5) es un dato
para demostrar (d 1) hay que demostrar (d 3)
para demostrar (d 3) hay que demostrar ((x 1) (b 2))
para demostrar (b 2) hay que demostrar (b 7)
(b 7) es un dato
para demostrar (x 1) hay que demostrar (b 3)
(b 3) es un dato
para demostrar (b 1) hay que demostrar (b 8)
(b 8) es un dato
Fin de la demostracion
```

4.4.- *Ejemplo.*

Cláusulas:

Las mismas que en 4.2.

Datos: (B)

Pretendida conclusion: A

Salida del programa:

```
para demostrar (a 1) hay que demostrar (a 2)
para demostrar (a 2) hay que demostrar ((g 1) (d 1));
no puedo hacer la demostracion
```

4.5.- *Ejemplo.*

Cláusulas:

$A \wedge B \wedge C \rightarrow D$

$I \wedge H \rightarrow B$

$H \wedge F \rightarrow B$

$A \rightarrow I$

$E \wedge F \rightarrow D$

$A \rightarrow F$

$K \wedge L \rightarrow E$

$A \rightarrow L$

Datos: (A,K)

Pretendida conclusión: D

Salida del programa:

```
para demostrar (d 1) hay que demostrar (d 3)
para demostrar (d 3) hay que demostrar ((e 1) (f 1))
para demostrar (f 1) hay que demostrar (a 2)
(a 2) es un dato
para demostrar (e 1) hay que demostrar (e 2)
para demostrar (e 2) hay que demostrar ((k 1) (l 1))
para demostrar (l 1) hay que demostrar (a 5)
(a 5) es un dato
para demostrar (k 1) hay que demostrar (k 2)
(k 2) es un dato
Fin de la demostracion
```

4.6.- *Ejemplo.*

Cláusulas:

Las mismas que en 4.5.

Datos: (A,K)

Pretendida conclusión: B

Salida del programa:

```
para demostrar (b 1) hay que demostrar (b 2)
para demostrar (b 2) hay que demostrar ((i 1) (h 1))
no puedo hacer la demostracion
```

4.7.- *Ejemplo.*

Cláusulas:

Las mismas que en 4.5.

Datos: {A,C,H}

Pretendida conclusión: D

Salida del programa:

```
para demostrar (d 1) hay que demostrar (d 2)
para demostrar (d 2) hay que demostrar ((a 1) (b 1) (c 1))
para demostrar (c 1) hay que demostrar (c 2)
(c 2) es un dato
para demostrar (b 1) hay que demostrar (b 2)
para demostrar (b 2) hay que demostrar ((i 1) (h 1))
para demostrar (h 1) hay que demostrar (h 4)
(h 4) es un dato
para demostrar (i 1) hay que demostrar (a 3)
(a 3) es un dato
para demostrar (a 1) hay que demostrar (a 6)
(a 6) es un dato
Fin de la demostracion
```

4.8.- *Ejemplo.*

Cláusulas:

Las mismas que en 4.5.

Datos: {A,C,H}

Pretendida conclusión: E

Salida del programa:

```
para demostrar (e 1) hay que demostrar (e 2)
para demostrar (e 2) hay que demostrar ((k 1) (j 1))
no puedo hacer la demostracion
```

4.9.- *Ejemplo.*

Cláusulas:

$B \rightarrow A$

$O \wedge R \rightarrow B$

$J \wedge K \rightarrow C$

$I \rightarrow D$

$J \rightarrow G$

$L \rightarrow K$

$L \wedge M \rightarrow I$

$R \wedge S \rightarrow N$

$U \rightarrow L$

$V \wedge X \rightarrow M$

$Z \wedge S \rightarrow Q$

$C \wedge D \wedge E \rightarrow A$

$G \wedge C \rightarrow B$

$G \wedge F \wedge H \rightarrow D$

$H \wedge I \rightarrow E$

$N \wedge L \rightarrow G$

$F \wedge H \rightarrow K$

$O \wedge P \wedge Q \rightarrow N$

$S \rightarrow L$

$U \rightarrow M$

$Y \wedge Z \rightarrow P$

$Z \wedge V \rightarrow U$

Datos: {J,V,Z}

Pretendida conclusión: A

```
para demostrar (a 1) hay que demostrar (b 2)
para demostrar (b 2) hay que demostrar ((g 1) (c 2))
para demostrar (c 2) hay que demostrar (c 4)
para demostrar (c 4) hay que demostrar ((j 3) (k 2))
para demostrar (k 2) hay que demostrar (l 8)
para demostrar (l 8) hay que demostrar ((u 8))
para demostrar (u 8) hay que demostrar (u 10)
para demostrar (u 10) hay que demostrar ((z 12) (v 10))
para demostrar (v 10) hay que demostrar (v 11)
(v 11) es un dato
para demostrar (z 12) hay que demostrar (z 13)
(z 13) es un dato
para demostrar (j 3) hay que demostrar (j 5)
(j 5) es un dato
para demostrar (g 1) hay que demostrar (j 2)
(j 2) es un dato
Fin de la demostracion
```

4.10.- *Ejemplo.*

Cláusulas:

Las mismas que 4.9.

Datos: {O,R}

Pretendida conclusión: A

Salida del programa:

```
para demostrar (a 1) hay que demostrar (b 1)
para demostrar (b 1) hay que demostrar ((o 1) (r 1))
para demostrar (r 1) hay que demostrar (r 6)
(r 6) es un dato
para demostrar (o 1) hay que demostrar (o 6)
(o 6) es un dato
Fin de la demostracion
```

CONCLUSIONES

Hemos presentado una formalización de conceptos como *juegos*, *grafos*, *árboles*, y procedimentos *min-max y alfa-beta*, hemos estudiado las relaciones de estos conceptos con los grafos equivalentes a problemas deductivos, y finalmente, hemos aplicado la estrategia α-β a la resolución de estos problemas, todo ello con el alcance y las limitaciones que se han visto.

Creemos que la memoria tiene tres aspectos de fundamental interés: en primer lugar, la formalización (por su ausencia en la literatura); en segundo, la demostración rigurosa de las relaciones entre juegos y procesos deductivos (por su interés científico intrínseco, y por presentar como análogas dos estructuras que aparecían como diferentes); y en tercero, la aplicación de α-β a la deducción (por su originalidad, y por su presumible ahorro de tiempo y memoria de cómputo).

Como hemos dicho, se presenta un trabajo académico y no un producto comercial; así, una posible continuación del trabajo consistiría en resolver los numerosos problemas de puesta a punto que un programa elaborado presenta: estudio y reducción de la complejidad, subprogramas que facilitasen la entrada y salida de datos, etc. Por ejemplo, es absolutamente innecesario que el programa nos diga,

"para demostrar (v 45) hay que demostrar (v 46)"

y, en realidad, toda referencia a la numeración de los nodos. Sin embargo, aquí hemos preferido dejarlo así, pues lo que se pretende es comprobar la corrección de los algoritmos, y la numeración resulta ser un objetivo esencial en las transformaciones a que son sometidas las estructuras que intervienen en el proceso.

En este orden de cosas, podría hacerse un estudio comparativo de la α-β deducción con otros métodos de razonamiento automático, en relación a tiempos de proceso y consumo de memoria; la dificultad ahora está en que habría que disponer de las realizaciones de estos métodos, para muchos de los cuales sólo se conoce su descripción. Sobre esto, cabe decir que en [PA] se muestra un estudio de este tipo entre min-max y α-β, y se viene a probar que con los mismos recursos, α-β permite una profundidad de exploración doble de la que permite min-max.

Nuestra modesta experiencia con programas de deducción nos hace creer que la realización, esto es, el programa que presentamos es bastante eficaz. De hecho, cuando preparábamos los ejemplos (sobre todo los últimos, con 22 reglas), vimos que el programa nos daba demostraciones distintas (y más cortas, claro) que la que habíamos preparado: realmente, el grafo correspondiente a estos ejemplos es bastante complicado de dibujar con claridad. Añadiremos un detalle: cuando no se puede hacer la demostración, el programa elige la "no demostración" más corta, esto es, se selecciona el subárbol más pequeño que termina en nodos perdedores.

Este trabajo se inscribe en la actividad investigadora del Grupo de Lógica de la Facultad de Matemáticas de Sevilla, una de cuyas líneas de trabajo es precisamente el Razonamiento Automático. En este contexto, el trabajo tiene una continuación clara: la extensión de estos resultados a otras lógicas. Sobre la Lógica de Primer Orden hemos hablado en IV.2.4; podría ser un buen punto intermedio el caso de la Lógica Monádica, pues permitiría interesarse en la extensión más que en el problema de unificación; otra posibilidad interesante está en el Intuicionismo: esta lógica, permite razonamientos hacia atrás (con sus reglas propias), pero no es apta para el método de resolución (que es intrínsecamente clásico). Dicho esto, pensamos que el mayor interés puede estar en las lógicas multivalentes y/o en la Lógica difusa.

Estas lógicas, de cada vez mayor presencia en los sistemas expertos, son aparentemente las más indicadas para la consideración de funciones heurísticas. De hecho, el comentario de la nota IV.3.1 apunta en este sentido; nuestra asignación de valores 1 y -1 para los nodos terminales no es más que una simplificación trivial.

BIBLIOGRAFIA:

[AAD] ADELSON-VELSKY, G.M., ARLAZAROV, V.L., DONSKOY, M.V. *Algorithms for Games*. Springer-Verlag, 1.988.

[BA] BANERJI, R.B. *Artificial Intelligence: A Theoretical Approach*. North-Holland, 1.980.

[BF] BARR, A., FEIGENBAUM, E.A. *The Handbook of Artificial Intelligence*. Vols. I y II. Kaufmann, 1.981.

[BO] BONNET, A. *Artificial Intelligence: Promise and Performance*. Prentice Hall, 1.985.

[CL] CHANG, C-1., LEE, R. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1.973.

[DE] DELAHAYE, J.P. *Outils logiques pour l'Intelligence Artificielle*. Eyrolles, 1.986.

[FA] FARRENY, H. *Exercices Programmés D'Intelligence Artificielle*. Masson, 1.987.

[FG] FARRENY, H., GHALLAB, M. *Eléments D'Intelligence Artificielle*. Hermes, 1.987.

[HA] HASEMER, T. LISP: *Une introduction à la programmation sur micro-ordinateur*. Addison-Wesley, 1.984.

[KN] KNUTH, D.E. *Algoritmos Fundamentales*. Reverté, 1.980.

[LE1] LEVY, D.N.L. *Computer Games I*. Springer-Verlag, 1.987.

[LE2] LEVY, D.N.L. *Computer Games II*. Springer-Verlag, 1.987.

[NI1] NILSSON, N.J. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1.971.

[NI2] NILSSON, N.J. *Principles of Artificial Intelligence*. Springer-Verlag, 1.982.

[PA] PAZOS, J. *Inteligencia Artificial*. Paraninfo, 1.987.

[RI] RICH, E. *Artificial Intelligence*. McGraw-Hill, 1.983.

[ST] SHIRAI, Y., TSUJII, J.-I. *Intelligence Artificielle: Concepts, Techniques et applications*. Eyrolles, 1.982.

[TZ] TZENG, C.-H. *A Theory of Heuristic Information in Game-Tree Search*. Springer-Verlag, 1.988.

[WI] WINSTON, P.H. *Artificial Intelligence*. Addison-Wesley, 1.984.