Inteligencia Artificial
Departamento de Lenguajes y Sistemas Informáticos
UPC - FIB

IA – Algoritmos de Juegos

Pau A.F.
15 / 06 / 2008

CONTENIDO

1. Introducción al Documento

1.1 Propósito

Este documento proporciona una visión divulgativa sobre el área de los algoritmos de juegos de la Inteligencia Artificial.

Su propósito es servir como punto de partida a quien desee introducirse en el área de los algoritmos de juegos con adversario y conocer sus elementos y características principales.

1.2 Visión General

El resto del documento contiene una definición de los algoritmos de juegos con adversario, cómo suelen plantearse estos problemas, y acto seguido toda una serie de algoritmos y técnicas utilizadas para mejorarlos (centrando nuestro peso en MiniMax).

Finalmente el lector puede disponer de unos cuantos ejemplos de áreas relacionadas donde estos algoritmos pueden (o son) utilizados, y sus aplicaciones más immediatas.

El último punto del documento, como bien indica su nombre, es la lista de enlaces web utilizados como bibliografía o que puedan resultar de interés.

2. Pongámonos en Situación

2.1 Qué es un Problema

La primera necesidad es definir el término problema. Sabemos lo que es un problema matemático, un problema económico, o el término problema en su mayor definición. Pero, ¿conocemos la definición de "problema" enfocada al mundo de la inteligencia artificial?

Un **problema**, en nuestro contexto, será la abstracción de una serie de elementos tales como: un objetivo, meta, o estado final a cumplir; un punto de inicio donde empezaremos a enfocar el problema; y una serie de movimientos que nos permitirán como mínimo aproximarnos del estado inicial al final. Y en el mejor de los casos, nos permitirán salir airosos con la mejor solución del problema.

Evidentemente, existen muchos tipos diferentes de problemas, pero todos ellos tienen elementos comunes que nos permiten clasificarlos, estructuralos, y afrontarlos automáticamente de un u otro modo según su tipo. Así pues, este documento se centrará en un único tipo de problema: los juegos.

Los problemas de juegos son aquellos en que varios agentes –o adversarios– compiten por lograr un mismo objetivo. Estos problemas se resuelven mediante los denominados "algoritmos de juegos", los cuales trataremos en gran profundidad más adelante.

No es difícil ver que dentro del conjunto de problemas con adversario están todos aquellos juegos de mesa para dos que seguramente todos hemos jugado (tres en raya, dominó, ajedrez, etc.). Pero no sólo esos juegos resolverán nuestros algoritmos, sinó que además podremos afrontas problemas de cualquier ámbito donde varios individuos compitan, ya sean juegos de cartas como el póker o el black jack, o incluso problemas del mundo real.

2.2 Tipos de Juegos

Los juegos, que ya de por si son una subcategoría de problemas, también pueden subclasificarse.

Incluso para los ordenadores no es lo mismo si intentas decidir la mejor jugada en el tres en raya que si pretendes decidir si jugando a cartas apuestas o te plantas. Por eso los juegos también deberán clasificarse según ciertas propiedades presentes en todos ellos, facilitando así la decisión de qué algoritmo utilizar para vencer.

La primera de las propiedades a tener en cuenta será **el número de jugadores** o agentes involucrados, información de gran vitalidad a la hora

de diseñar el algoritmo. Un juego puede ser sin adversario (por ejemplo un 8-puzzle), con 1 adversario (por ejemplo el 3 en raya), o con N adversarios.

La siguiente propiedad a conocer es **el orden de los movimientos**. Saber si por ejemplo los jugadores mueven alternativamente o por azar también es muy importante.

Una vez situados los jugadores y sus turnos, hay que saber qué conocimiento tienen éstos. En los juegos los jugadores pueden tener o bien **conocimiento perfecto**, donde no afecta el azar y todos saben en todo momento lo mismo, o bien **conocimiento imperfecto**, donde el azar sí puede participar y además hay ocultación de información entre jugadores.

Al hecho de que el azar aparezca o no aparezca en alguna de las propiedades anteriores se le conoce como **determinismo**.

También podemos encontrarnos juegos donde un movimiento que te beneficia siempre perjudica al mismo nivel al adversario (**equilibrio Nash**), o juegos donde ésto no ocurre.

Según sus características encontraremos juegos simétricos y asimétricos, de suma cero y de suma no cero, cooperativos, simultáneos y secuenciales, de información perfecta, de longitud infinita, o juegos bayesianos entre otros.

2.3 Estrategias a Seguir

La idea básica cuando buscas vencer computacionalmente un adversario es intentar predecir todos los posibles movimientos, todas las posibles situaciones desde tu turno hasta el final de la partida, y elegir la que prometa mejores resultados. Esta idea se la conoce como la de "generar todo el árbol de búsqueda". Pero esta estrategia tan bruta la gran mayoría de veces será algo imposible e inaceptable, pues una partida no puede durar siglos.

Lo que realmente haremos será buscar **una aproximación de las mejores jugadas**, guiándonos mediante heurísticos, y prediciendo únicamente las situaciones con mejor pinta (pues por ejemplo no sirve de nada saber las situaciones que podrían darse después de hacer un paso en falso).

Para poder realizar los cálculos computacionalmente, hará falta una abstracción codificable del juego. Entender e implementar realmente el problema.

Necesitaremos una representación del **estado** (válida para cualquier estadp), otra del estado inicial (desde dónde se empieza a jugar y quién inicia el juego), otra del estado ganador (ya sea por estructura o por propiedades), y finalmente definir los **operadores de movimiento** válidos de los jugadores, que determinarán las jugadas que puede hacer un agente desde el estado actual.

Además deberemos definir la **aproximación heurística**, es decir, la función que nos indique lo cerca que estamos de una juagada ganadora. En esta función intervendrá información del dominio del juego, y deberá tener un coste computacional lo más bajo posible.

Una vez definido el universo del juego, nos tocará aplicar el algoritmo más adecuado.

3. Métodos Utilizados en Juegos Sin Adversario

Para algoritmos de juegos unipersonales (por ejemplo los puzzles o el solitario) suelen utilizarse los algoritmos más comunes de **búsqueda heurística**, aunque realmente las soluciones a un juego unipersonal pueden encontrarse con cualquier algoritmo de fuerza bruta sin heurística alguna, siempre y cuando se disponga del tiempo y memoria necesarios.

De entre los algoritmos de búsqueda heurística, suele utilizarse o bien el algoritmo A*, que se describe en esta sección, o bien el algoritmo IDA* (según los recursos del computador).

El **algoritmo A*** (Hart, 1968) implementa una **búsqueda primero el mejor** (*best-first*). En otras palabras, se basa en intentar encontrar todos los movimientos necesarios para llegar del estado inicial al estado final, usando preferiblemente aquellos movimientos que sean más favorables. Así, entre otras cosas, se puede lograr minimizar los pasos de la solución, o el tiempo de cálculo.

Para conocer qué movimiento es mejor se utiliza una función de evaluación estática formada descompuesta como:

$$f = g + h$$

Donde "f" será el valor numérico del movimiento estudiado, la función "g" es una función del coste de llegar del estado inicial al estado actual, y la función "h" es una estimación del coste de llegar desde el estado actual al estado final o objetivo. Lógicamente, "h" es una estimación porque no conocemos a priori el camino exacto hacia la solución.

El algoritmo A*, simplificado y con sintaxis imprecisa, viene a ser el siguiente:

```
Algoritmo A*
        Estados_pendientes.insertar(Estado_inicial);
        Actual = Estados_pendientes.primero();
        Mientras(no_es_final(Actual) y (no Estados_pendientes.vacio)) hacer
                Estados_pendientes.borrar_primero();
                Estados_ya_estudiados.insertar(Actual);
                Sig_movimientos = generar_sucesores(Actual);
                Sig_movimientos = eliminar_repetidos(Sig_movimientos,
                                                     Estados_ya_estudiados,
                                                        Estados_pendientes);
                Estados_pendientes.insertar(Sig_movimientos);
                Actual = Estados_pendientes.primero();
        fMientras
fAlgoritmo
```

4. Métodos Utilizados en Juegos Con Adversario

Pero la chicha de los algoritmos de juegos se encuentra en los juegos con adversario. Aquí es donde estos algoritmo se diferencian de los demás algoritmos de búsqueda típicos.

A continuación se presentan los más importantes y las técnicas de mejora también más conocidas. Aún así existen muchísimos algoritmos más que quedaron fuera de este documento, como podrían ser DUAL*, Mem*, o B*.

4.1 Algoritmo MiniMax

El algoritmo **MiniMax** es el algoritmo más conocido (y utilizado) para problemas con exactamente **2 adversarios**, movimientos alternos ("ahora tú ahora yo"), e información perfecta. Además tiene un nombre la mar de ilustrativo, como veréis en los siguientes párrafos.

Identificaremos a cada jugador como **el jugador MAX** y **el jugador MIN**. MAX será el jugador que inicia el juego, el cual supondremos que somos nosotros, y nos marcaremos como objetivo encontrar el conjunto de movimientos que proporcionen la victoria a MAX (nosotros) independientemente de lo que haga el jugador MIN.

Etiquetar los jugadores como máximos y mínimos plasma perfectamente la dualidad de los objetivos de cada uno, además de que así se puede contabilizar perfectamente cómo una situación que mejora para un jugador hace que empeore para el otro. Matizar que no importa qué jugador deseemos ser, pues bajo nuestro punto de vista siempre calcularemos jugadas positivas a nuestro favor y negativas para el oponente.

Y todo ésto nos lleva a que por convenio una jugada vencedora del jugador MAX (lo que vendría a ser un "jaque mate") tendrá valor infinito, y una jugada del jugador MIN tendrá valor menos infinito.

Deberá existir para el juego a resolver una **función de evaluación heurística** que devuelva valores elevados para indicar buenas situaciones, y valores negativos para indicar situaciones favorables al oponente. Ahora ya podemos ver como para cada movimiento conoceremos su valor numérico en función de su "grado de bondad", y podremos ser capaces de identificar nuestro mejor movimiento.

Además de utilizar la anterior función heurística, usaremos una estrategia DFS de profundidad limitada para explorar el conjunto de jugadas. Como es imposible hacer una exploración exhaustiva de todas las jugadas, se hace una búsqueda limitada en profunidad. Significa que en lugar de estudiar todas las posibles situaciones hasta el fin de la partida, se buscarán por ejemplo todas las situaciones hasta de aquí 3 turnos. Aunque ésto limitará lo que veamos del espacio de búsqueda, y quizás nos incitará a realizar unas jugadas que a la larga sean nefastas. La calidad de nuestras jugadas

vendrá determinada por la profundidad a la que lleguemos en cada exploración. Cuanto más profunda sea, mejor jugaremos, pero mayor coste tendrá el algoritmo.

Por lo tanto, situados todos los elementos del algoritmo, veamos cómo es MiniMax:

```
Función MiniMax(estado) retorna movimiento
        mejor_mov: movimiento;
        max, max_actual: entero;
        max = -infinito;
        para cada mov en movimientos_posibles(estado) hacer
                max_actual = valorMin(aplicar(mov, estado);
                si max_actual > max entonces
                        max = max_actual;
                        mejor_mov = mov;
                fsi
        fpara
        retorna mejor_mov;
fFunción

Función valorMax(estado) retorna entero
        valor_max:entero;
        si estado_terminal(estado) entonces
                retorna evaluación(estado);
        sinó
                valor_max := -infinito;
                para cada mov en movimientos_posibles(estado) hacer
                        valor_max := max(valor_max, valorMin(aplicar(mov, estado));
                fpara
                retorna valor_max;
        fsi
fFunción

Función valorMin(estado) retorna entero
        valor_min:entero;
        si estado_terminal(estado) entonces
                retorna evaluación(estado);
        sinó
                valor_min := +infinito;
                para cada mov en movimientos_posibles(estado) hacer
                        valor_min := min(valor_min, valorMax(aplicar(mov, estado));
                fpara
                retorna valor_min;
        fsi
fFunción
```

El algoritmo tiene una función principal (MiniMax) que será la que usemos, y la que nos devolverá la mejor realizable, y dos funciones recursivas mutuas (valorMax y valorMin) que determinan el valor de las jugadas dependiendo de si es el turno de MAX o de MIN. De este modo cada nivel elegirá el valor que propaga a su ascendiente dependiendo de qué jugador es. En los niveles de MAX se elegirá el valor mayor de todos los descendientes, y en los niveles de MIN se elegirá el menor. Tras esto la

máquina supone que todo jugador elige siempre su mejor jugada, cosa que no siempre será.

La función "estado_terminal" determina si el estado actual es una solución o pertenece al nivel máximo de exploración. La función "evaluación" calcula el valor heurístico del estado.

Pero esta versión, aunque sea sencilla, no es la más inteligente a la hora de tratar juegos bipersonales. Por eso al algoritmo MiniMax se le pueden efectuar una serie de técnicas para mejorarlo, buscando siempre una decisión más segura del movimiento, y un tiempo de cálculo menor.

4.2 Técnicas para Mejorar MiniMax

Pero MiniMax puede ser mucho más inteligente de lo anteriormente expuesto. Veamos qué tipo de mejoras pueden aplicársele.

4.2.1 Poda Alfa-Beta

Un punto clave que hemos de tener en cuenta es que cuanto más profunda sea la exploración, mejor podremos tomar la decisión sobre qué jugada debemos escoger.

Para juegos con un factor de ramificación elevado, esta profundidad no podrá ser muy grande, ya que el cálculo necesario para cada decisión será prohibitivo. Su tiempo de exploración será prohibitivo.

Para reducir este tiempo se han estudiado los árboles de búsqueda, y se ha llegado a la conclusión de que es muy factible usar heurísticos con poda. En otras palabras, usaremos una técnica de **ramificación y poda** (*branch and bound*) con la cual una solución parcial se abandonará cuando se compruebe que es peor que otra solución ya conocida.

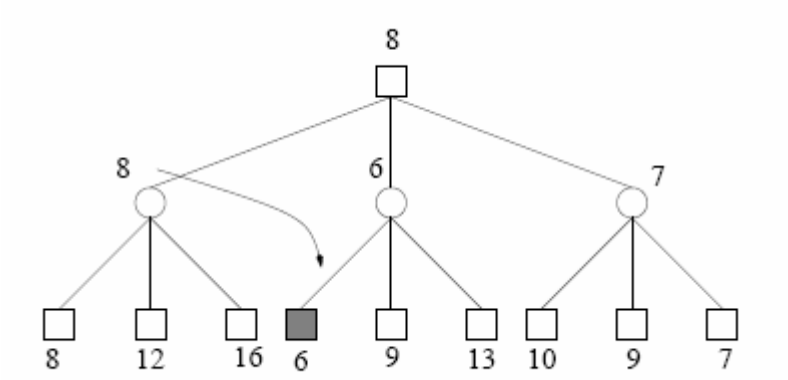
La estrategia denominada alfa-beta aprovecha que el algoritmo del MiniMax hace una exploración en profundidad para guardar información sobre cuál es el valor de las mejores jugadas encontradas para cada jugador.

Concretamente guarda dos valores, denominados **umbrales alfa y beta** (de ahí el nombre del heurístico).

El valor alfa representa la cota inferior del valor que puede asignarse en último término a un nodo maximizante, y análogamente el valor beta representará la cota superior del valor que puede asignarse como última opción en un nodo minimizante.

De esta manera el algoritmo podrá acotar el valor de una exploración, pues gracias a los umbrales alfa y beta conocerá si vale la pena explorar un camino o no.

Veamos un ejemplo:



En el árbol superior podemos ver que el valor que propagaría MiniMax al nodo raíz sería 8. Pero si nos fijamos en el nodo marcado, en el segundo descendiente de la raíz ya sabe que el mejor nodo del primer descendiente tiene valor 8. Puesto que el desdendiente es un nodo Min, y se ha encontrado por debajo un valor de 6, el algoritmo interpretará que por ese camino como mejor resultado habrá ese 6 (pues el nodo inferior elegirá o el 6 o algo inferior). Como ya conocemos el posible 8, todos los nodos que cuelgan del Min que elegiría el 6 ya no hace falta explorarlos, pues seguro que ahí no se encuentra nuestra mejor jugada. En este momento se realizaría la poda de esa rama e iría a explorar el tercer posible camino.

Podemos observar también que esta circunstancia se repetirá en el tercer descendiente al explorar su último nodo, pero al no quedar más descendientes la poda no nos ahorraría nada. Significa que la efectividad de esta heurística dependerá del orden de los nodos, pues cuanto antes encuentres un valor penalizado por el umbral, antes podrás podar la rama.

Esta heurística modifica el algoritmo del minimax introduciendo dos parámetros en las llamadas de las funciones, alfa y beta, y representarán el límite del valor que pueden tomar las jugadas que exploramos. Una vez superado ese límite sabremos que podemos parar la exploración porque ya tenemos el valor de la mejor jugada accesible.

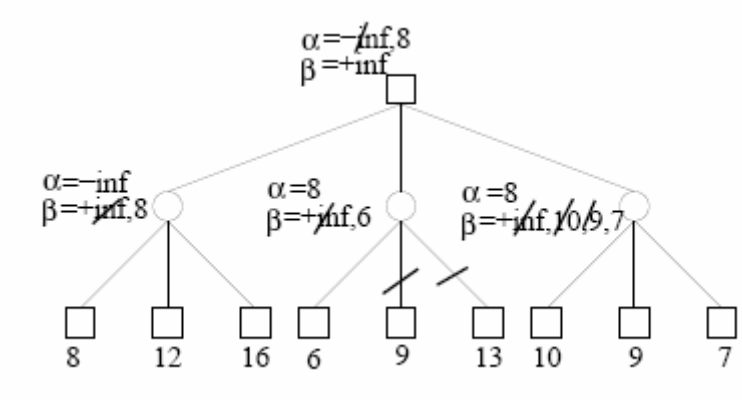
La función MiniMax se mantiene igual. Sólo varían las funciones que hacen la exploración de cada nivel, el código de las cuales sería el siguiente:

```
funcion valorMax( g , alfa , beta ) retorna entero
        si estado_terminal( g ) entonces
                retorna ( evaluacion( g ) )
        sino
                para cada mov en movs_posibles( g ) hacer
                        alfa:=max( alfa , valorMin( aplicar(mov, g ), alfa, beta ) )
                        si alfa >= beta entonces
                                retorna ( beta )
                        fsi
                fpara
                retorna ( alfa )
        fsi
ffuncion
```

```
funcion valorMin( g , alfa , beta ) retorna entero
        si estado_terminal( g ) entonces
                retorna ( evaluacion( g ) )
        sino
                para cada mov en movs_posibles( g ) hacer
                        beta:=min( beta , valorMax( aplicar (mov, g ), alfa, beta ) );
                        si alfa >= beta entonces
                                retorna ( alfa )
                        fsi
                fpara
                retorna ( beta )
        fsi
ffuncion
```

La llamada que hará la función MiniMax a la función valorMax le pasará −infinito como valor de alfa y +infinito como valor de beta (pues al no conocerse ningún valor es imposible marcar un umbral predefinido).

En la siguiente figura se ve cómo exploraría el algoritmo de MiniMax con poda alfa beta el ejemplo anterior:



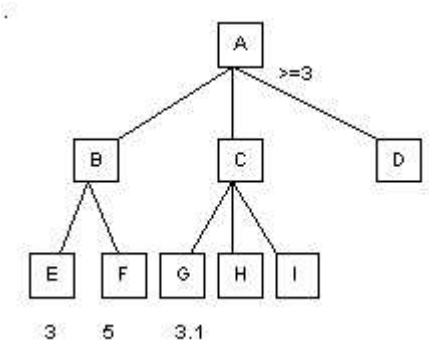
En colcusión, con poda alfa beta podremos ampliar el límite de profundidad de nuestra exploración, ya que con las podas nos ahorramos la exploración de varias parte del árbol de búsqueda, llegándonos a ahorrar así múltiples niveles. De ahí deriva que algunas de las siguientes ampliaciones se basen en mejorar el comportamiento de la poda alfa-beta.

4.2.2 Poda de Inutilidades

Realmente la poda de inutilidades es un refinamiento adicional de la poda alfa-beta.

La **poda de inutilidades** es la finalización de la exploración de un subárbol que ofrece pocas posibilidades de mejora sobre otros caminos que ya han sido explorados.

Es decir, es la misma idea de alfa-beta, pero en lugar de podar una rama si sabes que no tendrás resultados mejores, **podarás si sabes que no tendrás resultados suficientemente mejores**.



En el ejemplo de la figura, después de examinar el nodo B, el jugador maximizante se asegura un valor de 3. Al examinar el nodo G se obtiene un valor 3.1 que es sólo algo mejor que 3. Puesto que si se exploran los nodos H e I no lograremos mejorar este valor desde el punto de vista del jugador maximizante, sería mejor podarlos y dedicar más tiempo a explorar otras partes del árbol donde se puedan obtener más ventajas.

4.2.3 Espera del Reposo

Cuando la condición de corte del algoritmo MINIMAX está basada sólo en una profundidad fija, puede darse el llamado **efecto horizonte**. Este efecto ocurre cuando se evalúa como buena o mala una posición sin saber que en la siguiente jugada la situación se revierte.

La **espera del reposo** busca evitar el efecto horizonte.

Una condición adicional de corte de recursión en el algoritmo MiniMax debería ser el alcanzar una situación estable. Si se evalúa un nodo del árbol de juego y este valor cambia drásticamente al evaluar el nodo después de realizar la exploración de un nivel más, la búsqueda debería continuar hasta que ésto dejara de ocurrir.

A esta actuación se la conoce como esperar el reposo y nos asegura que las medidas a corto plazo no influyan idebidamente en la elección, problema siempre presente por no explorar exhaustivamente las jugadas.

Un ejemplo donde podría presentarse el efecto horizonte es en un intercambio de piezas, donde a cada turno el jugador logra intercambiar y así pues la evaluación cambia drásticamente de un lado hacia otro.

4.2.4 Búsqueda Secundaria

Otra forma de mejorar el procedimiento MiniMax es realizar **una comprobación del movimiento elegido**, asegurándonos que no haya ninguna trampa algunos movimientos más allá de los que exploró la búsqueda inicial.

La idea es muy sencilla. Una vez el procedimiento normal de MiniMax haya elegido un movimiento concreto explorando "n" niveles, la técnica de búsqueda secundaria consistirá en realizar una **búsqueda adicional** de "d" niveles en la única rama escogida para asegurar que la elección sea buena. Así si encuentra que más adelante todas las posibilidades empeoran, se podrá optar por elegir otra rama antes de caer directo en alguna trampa.

El algoritmo se vuelve más costoso, pero por otra parte la decisión es más robusta.

4.2.5 Uso de Movimientos de Libro

Para muchos juegos suelen existir un conjunto de **movimientos y jugadas prefijadas** que mucha gente use y sean de conocimiento general, aunque solo sea en ciertos momentos de la partida.

Así, sería satisfactorio conocer si en nuestro juego se da el caso, y si es así, seleccionar el movimiento idóneo consultando la configuración actual del tablero. Naturalmente, en juegos complicados, esta estrategia será imposible de realizar, pues la tabla sería exageradamente enorme y compleja.

Sin embargo, este enfoque resulta razonable (y útil) para algunas partes de ciertos juegos. Y en los momentos donde tablas no sepa encontrar un movimiento idóneo, su combinación con el uso de la búsqueda MiniMax mejorará el rendimiento del programa. Y sus resultados.

Por ejemplo, se pueden usar movimientos de tablas en aperturas y finales del ajedrez, pues están altamente estudiados y pueden ser catalogados.

4.2.6 Búsqueda Sesgada

Otra forma más de podar un árbol de juegos es hacer que **el factor de ramificación varíe** a fin de dirigir más esfuerzo hacia las jugadas más prometedoras.

Se puede realizar una búsqueda sesgada clasificando cada nodo hijo, quizás mediante un evaluador estático rápido. Es decir, que según los resultados que prometa un movimiento, se buscarán más o menos siguientes situaciones más allá de éste.

Así evitaremos perder tiempo recorriendo situaciones que a priori no destaquen por sus resultados.

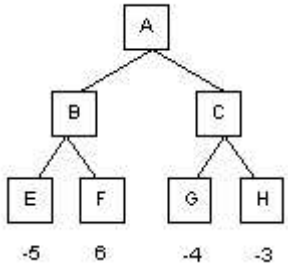
4.2.7 MiniMax Dependiente del Adversario

Según realiza la exploración del árbol de juego, el algoritmo MiniMax supone que el oponente siempre elige el camino óptimo.

Si se está frente a un adversario muy inteligente, éste podría llegar a explorar más capas que las exploradas por el algoritmo, y por lo tanto tomar otra decisión que la supuesta, pues el camino que tu no crees óptimo realmente a la larga lo sea para él.

Además, el adversario también podría equivocarse y no elegir el camino óptimo. La consecuencia es que MiniMax elegirá el movimiento basándose en la errada suposición del movimiento óptimo del adversario.

Ante una situación de derrota, según sugiere Berliner (1977), podría ser mejor asumir el riesgo de que **el oponente puede cometer un error**. Ilustremos la idea con un ejemplo.



En el ejemplo superior, se debe elegir entre dos movimientos B y C que conducen a la derrota siempre que el oponente elija el movimiento óptimo, pues tu eres A.

Según MiniMax, se debería elegir el movimiento menos malo de los dos, es decir, el C, con una valor de derrota -4. Suponga que el movimiento menos prometedor lleva a una situación muy buena para nosotros si el oponente comete un único error. En el ejemplo, el nodo B resulta el menos prometedor por tener un valor de -5, sin embargo, si el oponente comete un error elegirá F, el cual lleva a una situación muy ventajosa para nosotros. Sin embargo, en C aunque el oponente cometa un error acabaremos perdiendo. Dado que frente a un oponente que nunca falle no resulta mucho mejor una derrota que otra, se puede asumir el riesgo de que el oponente cometa un error ya que esto conduciría a una situación muy ventajosa. Así pues se elegiría el nodo B en lugar del C dictado por MiniMax básico.

Análogamente, si debe elegirse entre dos movimientos buenos, uno ligeramente mejor que el otro, podría resultar mejor elegir el menos mejor si al asumir el riesgo de que el oponente se equivoque nos conduce a situaciones más ventajosas.

Para tomar esta clase de decisiones correctamente, el algoritmo debería modelar el estilo de juego particular de cada oponente. Ésto permitiría estimar la probabilidad de que cometa distintos errores.

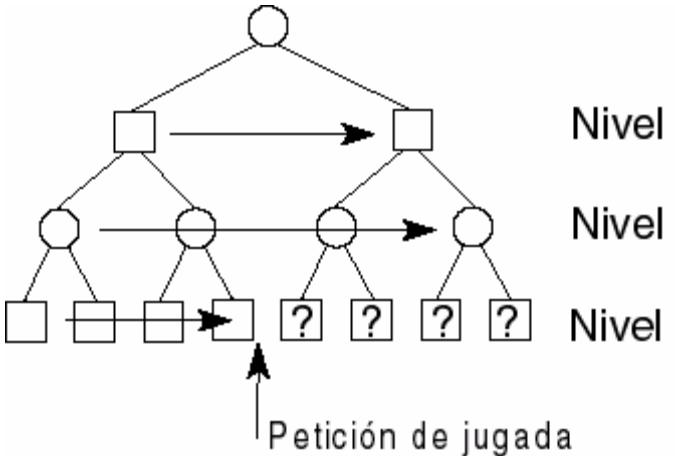
Sin lugar a dudas, esta ampliación es muy difícil de lograr y se necesita contar con **técnicas de aprendizaje** para que el algoritmo obtenga conocimiento sobre su oponente a lo largo del juego.

4.2.8 Técnica de Bajada Progresiva

Algunas veces la restricción que se nos impone durante el cálculo no es de memoria, sinó de tiempo. Debemos decidir nuestra próximo movimiento en un periodo máximo.

En estos casos, lo mejor sería aprovechar hasta el último segundo computacional disponible, y para ello se diseñó la técnica de **bajada progresiva**.

Esta técnica consite en ir recorriendo todos los nodos por niveles hasta que nos llegue la petición de jugada (fin del tiempo), y entonces devolveremos la solución encontrada en el último nivel procesado por completo.



4.2.9 Continuación Heurística

La técnica de **Continuación Heurística** intenta evitar el peligroso efecto horizonte, provocado por la limitación en profundidad. Es similar a la técnica de "Búsqueda Secundaria", pero no es el mismo concepto.

En la Continuación Heurística, se realiza previamente una búsqueda en anchura del árbol hasta el nivel de profundidad seleccionado. Una vez realizada la búsqueda, en lugar de elegir el mejor movimiento, aquí será donde añadiremos la modificación: seleccionaremos **un conjunto de nodos terminales** y los desarrollaremos a mayor profundidad.

Esta selección de terminales a "asegurar" vendrá dada por un conjunto de heurísticas directamente relacionadas con el juego.

Por ejemplo, podría calcularse si en ajedrez en un terminal tienes un rey en peligro o un peón a punto de convertirse en dama, y entonces para estas situaciones críticas buscar a mayor profundidad.

4.2.10 Movimiento Nulo

El Movimiento Nulo, o *Null-move forward pruning*, fue presentado por Chrilly Donniger en 1993.

Esta técnica permite reducir de forma drástica el factor de ramificación con un cierto **riesgo de perder información importante**.

La idea es dar al oponente **una jugada de ventaja**, y si tu posición sigue siendo buena, (alfa mayor que beta), se asumirá que el alfa real seguirá siendo mayor que beta, y por tanto podamos esa rama y seguimos examinando otros nodos.

Aún así esta técnica, a parte del riesgo de perder información, tiene otros dos inconvenientes: inestabilidad en la búsqueda (pues los valores de beta pueden cambiar), y que no se suelen permitir dos movimientos nulos seguidos.

```
/* el factor de reducción de profundidad */
#define R 2
int search (alpha, beta, depth) {
   if (depth <= 0)
      return evaluate();
   /* miramos si es legal y deseado hacer un movimiento nulo */
   if (!in_check() && null_ok()) {
      make_null_move();
      /* movimiento nulo con ventana mínima alrededor de beta */
      value = -search(-beta, -beta + 1, depth - R - 1);
      if (value >= beta) /* podar en caso de superar beta */
         return value;
   }
   /* continuar con búsqueda NegaScout/PVS  */
   . . .
}
```

Y como siempre, pues pueden encontrarse distintas versiones del Movimiento Nulo, como por ejemplo el Verificado (*Verified Null-move forward pruning*), que sustituye la poda del movimiento nulo por la realización de una búsqueda a poca profundidad.

4.2.11 Aspiration search

Esta mejora es simplemente **una mejora en la llamada** a Alfa-Beta. Normalmente, el nivel superior hará una llamada del tipo:

```
AlfaBeta(pos, depth, -infinito, +infinito) ;
```

Usando búsqueda por aspiración cambiaremos esta llamada por:

```
AlfaBeta(pos, depth, valor-ventana, valor+ventana) ;
```

Donde 'valor' es una aproximación del resultado esperado, y la ventana es la desviación que creemos podrá tener dicho valor.

Con esta llamada exploraremos menos nodos pues los límites alfa/beta ya se usarán desde buen principio. El peligro es que la búsqueda fallará si el valor no se encuentra dentro de nuestras suposiciones, en cuyo caso habrá que rehacer la búsqueda.

4.2.12 Algoritmo NegaMax

Finalmente presentar el algoritmo NegaMax, que simplemente es una **versión más compacta del MiniMax**.

En lugar de usar dos subrutinas auxiliares, una para el jugador Max y otra para el Min, simplemente usa la puntuación negada del siguiente aprovechando la siguiente relación matemática:

$$max(a, b) == -min(-a, -b)$$

Así pues, ésta sería la pinta aproximada que tendría el algoritmo de NegaMax:

```
funcion NegaMax( g ) retorna entero
        si ( estado_terminal( g ) ) entonces
                        retorna evalua( g )
        fsi
        max := -infinito
        para cada mov en movs_posibles( g ) hacer
                valor := -NegaMax( aplicar(mov, g))
                si valor > max entonces
                        max := valor
                fsi
        fpara
        retorna ( max )
ffuncion
```

Evidentemente a esta versión comprimida del MiniMax también puede aplicársele la poda alfa-beta.

4.2.13 Algoritmo NegaScout

El algoritmo NegaScout, evolución del NegaMax, todavía puede mejorar el rendimiento en juegos de ajedrez en un 10% estimado.

Como Alfa-Beta, NegaScout es un algoritmo de búsqueda direccional para calcular el valor minimax de un nodo en un árbol. La mejora es que NegaScout supera al algoritmo Alfa-Beta en el sentido que el primero jamás examinará un nodo que el segundo podaría, y además podaría otros más.

NegaScout tiene un comportamiento óptimo cuando los movimientos (nodos) están correctamente ordenados. Así producirá más podas que Alfa-

Beta al **asumir que el primer nodo explorado es el mejor**. En otras palabras, el primer nodo a explorar deberá ser el que ofrezca máxima ventaja al jugador actual. Entonces, dada esta suposición, comprobará si es cierto realizando una búsqueda en los otros nodos con una ventana nula (cuando alfa y beta son iguales), cosa que es más rápida que buscar con una ventana alfa-beta normal y corriente. Si la comprobación falla, significará que el primer nodo no era el de máxima variación (el de mayor ventaja), y la búsqueda proseguirá siendo una alfa-beta típica.

Claramente se aprecia por qué la máxima ventaja en NegaScout aparece si hay definido un buen orden en los movimientos. En cambio, con un orden aleatorio, NegaScout tardará más que Alfa-Beta, pues aunque no explore nodos podados por Alfa-Beta, deberá tratar varias veces los otros.

Su nombre proviene del hecho de aprovechar la relación matemática de negación explicada en el anterior apartado, y del uso de la ventana nula, pues también se la conoce como "*scout window*".

Su pseudocódigo es el siguiente:

```
funcion NegaScout( g , depth, α, β) retorna entero
        si ( estado_terminal( g ) ó  depth=0) entonces
                        retorna evalua( g )
        fsi
        b := β
        para cada mov en movs_posibles( g ) hacer
                valor := -NegaScout( aplicar(mov, g), depth-1, -b, -α )
                si α < valor < β y no-es-primer-nodo-hijo
                        valor := -NegaScout(aplicar(mov, g), depth-1, -β, -valor)
                fsi
                α := max(α, valor)
                si α ≥ β entonces
                        retorna (α)
                fsi
                b := α+1
        fpara
        retorna (α)
ffuncion
```

Hoy en día NegaScout todavía está considerado en la práctica el mejor algoritmo de búsqueda por muchos programadores de ajedrez. Y mucha gente también lo conoce como PVS (*Principal Variation Search*) o como una modificación de éste.

*4.3 Algoritmo SSS**

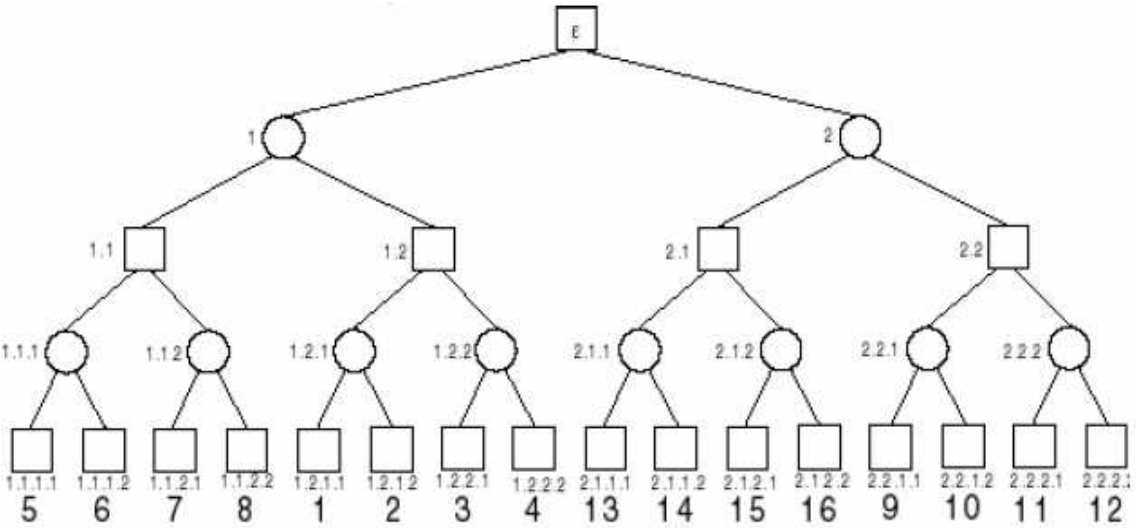
Diseñado en 1979 por George Stockman, el algoritmo SSS* introdujo un enfoque radicalmente distinto a la poda Alfa-Beta de búsqueda en árboles MiniMax de profundidad prefijada. Tiene sus similitudes con el algoritmo A*, pues explora caminos tipo *best-first*.

Este algoritmo es superior al MiniMax Alfa-Beta, pues poda los nodos que alfa-beta podaría, pero además no explora algunos nodos que alfa-beta sí exploraría (así pues delimita todavía más el espacio de búsqueda).

Esta mejora respecto al alfa-beta es debida a que el algoritmo SSS* utilitza subárboles del árbol de juego con los cuales solucionará el problema de **la ordenación en los nodos terminales**. Sin embargo, la estructura utilizada para manejar los nodos implica un aumento de la complejidad espacial, que es la desventaja de este algoritmo.

Estos subárboles tendrán un valor asignado, que será el límite inferior más alto de sus constituyentes. Este valor siempre será mayor o igual que la mejor solución encontrada dentro de ese subárbol, por lo cuál podremos usarlo para realizar la ordenación.

Además, en esta estructura SSS* utilizará la **notación decimal de Dewey** para representar los nodos en el árbol, donde el nodo raíz será "Epsilon", y el resto de nodos serán N.n (hijo n, desde la izquierda, del nodo N):



Otra característica de SSS* es que usa tripletas(N, s, h) para cada estado, donde 'N' es la numeración del nodo, 's' es el estado de la solución N, y 'h' es el valor del estado [-∞, +∞]. El estado 's' de la solución podrá ser etiquetado como VIVO, que indica que aún pueden generarse más sucesores del nodo, o SOLUCIONADO, que indicará que todos los sucesores ya fueron generados.

Ya introducidos en la notación, centrémonos en el funcionamiento y código del algoritmo.

Podríamos decir que SSS* funciona en dos etapas:

1. Generación de un conjunto de ramas, expandiendo el primer hijo si se trata de un nodo MIN y todos los hijos si se trata de un nodo MAX.

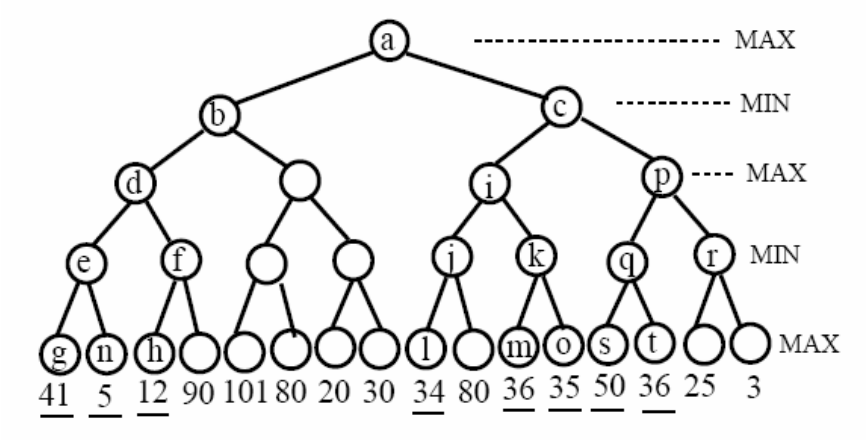
2. Selección de estados según su valor h, generando sucesores de los nodos no solucionados hasta que se solucionen. En el caso de que un

nodo esté solucionado se soluciona el padre (si es nodo MIN o el último hermano de un MAX), o se genera el siguiente hermano en el caso de tratarse de un nodo MAX que no sea último hermano.

Así pues, el algoritmo SSS* en pseudocódigo será el siguiente:

```
Algoritmo SSS* . V(N)
Entrada: Nodo N.
Salida: Valor minimax de dicho nodo.

Insertar en LISTA el estado inicial (N = ϵ, s = VIVO, h = +∞)
Hacer siempre
   Seleccionar y eliminar el primer estado de LISTA: p = (N, s, h).
   Si N = ϵ y s = SOLUCIONADO entonces devolver h FinSi
   Expandir el estado p de la siguiente forma:
   Si s = VIVO entonces
      Si N no es terminal entonces
         Si N es MAX entonces
            Para n = b hasta 1 hacer
               Insertar el estado (N.n, s, h) en la cabeza de LISTA.
            FinPara
         sino /*N es MIN*/
            Insertar el estado (N.1, s, h) en la cabeza de LISTA.
         FinSi
      sino /*N es terminal*/
         Insertar el estado (N, SOLUCIONADO, min{h, f(N)}) en LISTA, delante
de todos los estados con menor h que este estado.
      FinSi
   sino /*s = SOLUCIONADO*/
      Si N es MAX entonces
         Sea N = M.n
         Si n ≠ b entonces
            Insertar (M.n + 1, VIVO, h) en la cabeza de LISTA.
         sino /*n = b*/
            Insertar (M, SOLUCIONADO, h) en la cabeza de LISTA.
         FinSi
      sino /*NesMIN*/
         Sea N = M.n
         Insertar (M, SOLUCIONADO, h) en la cabeza de LISTA.
         Eliminar de LISTA todos aquellos estados sucesores de M.
      FinSi
   FinSi
FinHacer
```

La siguiente figura muestra un ejemplo de ejecución de SSS* donde las letras dentro de los nodos indican el orden en que se evalúan los nodos, y los números subrayados muestran los nodos terminales que son evaluados:



Posteriormente se han propuesto varias pequeñas variantes de SSS* como InterSSS*, RecSSS*, Dual* o Mem*, las cuales hacen un uso más eficiente de memoria y los recursos del sistema.

4.4 Algoritmo Scout

Desarrollado por Judea Pearl en los años 80, este algoritmo tiene como idea básica que mucho esfuerzo computacional se puede evitar si se tiene una forma rápida de **comprobar desigualdades**. De nuevo la potencia del algoritmo se deberá a su capacidad de poda: no explorará ramas que no lleven a una mejor solución. Además utilizará menos memoria que SSS*.

Por lo tanto con SCOUT necesitaremos, indispensablemente, un método rápido para comprobar desigualdades. Este método, que comprobará si el valor MiniMax de un nodo es menor o mayor que un determinado valor, lo denominaremos función TEST.

En consecuencia, para SCOUT necesitaremos **dos rutinas**: "TEST" y "EVAL".

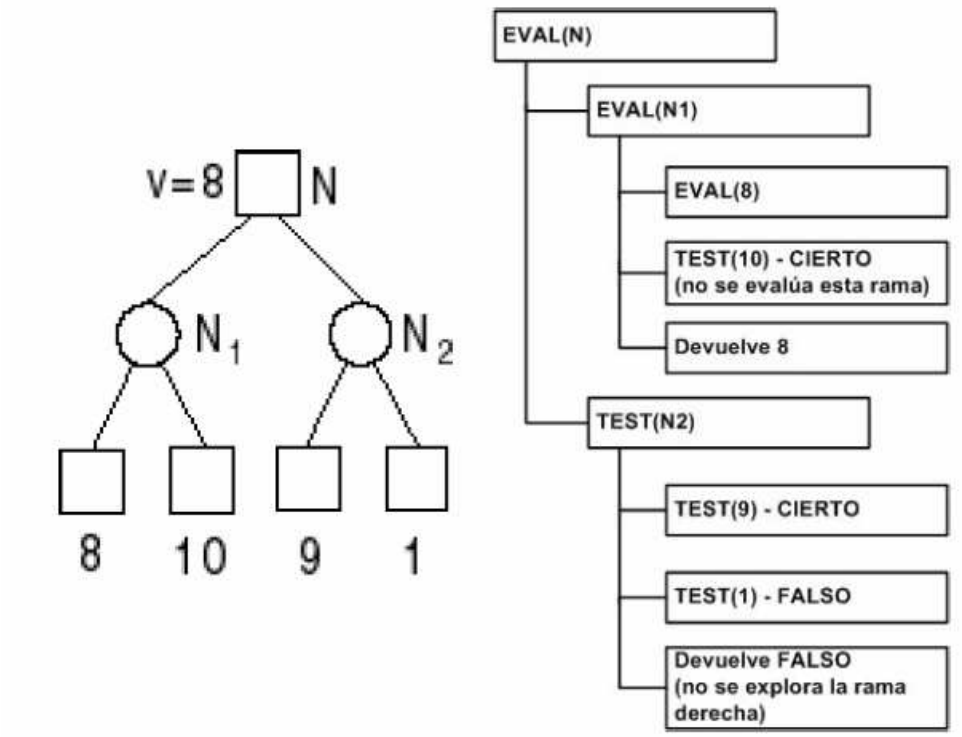
La función **TEST** se encargará de resolver la desigualdad. Sus parámetros de entrada serán: el nodo a aplicar el test, el valor fijo de la comparación, y la desigualdad a aplicar (> ó ≥) . Su salida será un booleano indicándonos si se cumple o no la desigualdad. Su pseudocódigo es el siguiente:

```
Algoritmo TEST. TEST(N,v,>)
Entrada: Nodo N, valor v, operando >.
Salida: Verdadero o falso.

Si N es nodo terminal entonces
        Si f(N)>v entonces devolver CIERTO
        sino devolver FALSO
        FinSi
sino
        Para i=1 hasta b hacer
                Si N es MAX entonces
                        Si TEST(Ni,v,>) entonces devolver CIERTO FinSi
                Sino /*N es MIN*/
                        Si NO(TEST(Ni,v,>)) entonces devolver FALSO FinSi
                FinSi
        FinPara
/* Ningún hijo cumple la condición */
Si N es MAX entonces devolver FALSO sino devolver CIERTO FinSi
FinSi
```

Por otra parte, la función **EVAL** calculará el valor MiniMax de un nodo, usando a TEST para comprobar cuándo deberá o no explorar una rama determinada. Su único parámetro de entrada será el nodo a evaluar.

```
Algoritmo EVAL. EVAL(N)
Entrada: Nodo N
Salida: Valor minimax de dicho nodo.

Generar los hijos de N:N1, N2, Nb
Si N es nodo terminal
        v=f(N)
sino
        v=EVAL(N1)
        Para i=2 hasta b hacer
                Si N es MAX entonces
                        Si TEST(Ni,v,>) entonces v=EVAL(Ni) FinSi
                sino /*N es MIN*/
                        Si NO(TEST(Ni,v,≥)) entonces v=EVAL(Ni) FinSi
                FinSi
        FinPara
FinSi
Devolver v.
```

Finalmente, presentadas ya las sencillas rutinas que conforman SCOUT, en la siguiente figura se puede observar el ejemplo típico de aplicación del algoritmo:



4.5 Algoritmo MTD(f)

MTD(f) es la abreviatura de MTD(n,f), que a su vez es la abreviatura de "*Memory-enhanced Test Driver node n and value f*". El algoritmo que ahora trataremos es un algoritmo de búsqueda en árboles minimax usando ventanas nulas y tablas de transposiciones (*transposition tables*).

Antes de entrar en materia con el algoritmo, veamos qué es una tabla de transposicion.

Una **tabla de transposición** es una base de datos donde se almacenan los resultados de búsquedas previamente realizadas. Es una forma de reducir drásticamente el espacio de búsqueda con poquísimo impacto negativo. Por ejemplo, en el juego de ajedrez, durante su búsqueda de fuerza bruta, pueden repetirse una vez tras otra las mismas posiciones, pero desde secuencias distintas de movimientos (estos "estados repetidos" son una "transposicion"). Cuando nos encontramos con una transposición, es beneficiosos recordar qué determinamos la última vez. Así ahorraremos repetir de nuevo toda la búsqueda.

Por esta razon, una tabla de transposiciones será una tabla de *hash* de gran capacidad en la que almacenaremos estados previamente buscados, hasta qué nivel se les realizó la búsqueda, y qué determinamos de éstos. Así pues su acceso tendrá un bajísimo coste constante, pues se accederá mediante hash keys derivadas del estado.

Con la tabla, cada vez que lleguemos a una nueva posición, el programa comprobará la tabla para ver si antes ya se ha analizado esa posición. Si está, se recoge el valor anteriormente asignado a esa posición y se usa directamente. Si no se encuentra, el valor se calculará con la función de evaluación y se insertará esa posición-profundidad-valor en la tabla, por si más adelante reaparece.

Una vez comprendido el concepto de "tablas de transposiciones", ya nos encontramos en situación de comprender fácilmente lo que hace el algoritmo MTD(f) en un código de apenas diez líneas.

En él se usa la función "*AlphaBetaWithMemory*", que no es más que una versión de Alfa-Beta que usa tablas de transposiciones. Esta función se encargará de encontrar el valor minimax de cierto nodo. Para buscar el valor necesario, MTD(f) usará esta función siempre con un tamaño de ventana igual a cero (**ventana nula**).

MTD(f) trabajará acercándose como si de un *zoom* se tratara al valor minimax buscado. Cada llamada a Alfa-Beta retornará un límite superior o inferior del rango donde se encuentra el valor deseado. Estos límites se almacenarán en *upperbound* y *lowerbound*, formando un intervalo alrededor del auténtico valor minimax buscado.

Al inicio de la búsqueda, como desconocemos completamente cuál es el valor buscado, los umbrales del intervalo serán más y menos infinito. En cambio, durante las iteraciones, cuando ambos umbrales colisionen (tengan el mismo valor) significará que ya hemos afinado tanto el intervalo que ya tendremos el valor minimax concreto.

Así pues, sin entrar en detalle en la implementación concreta de *AlphaBetaWithMemory*, el algoritmo MTD(f) será tan sencillo como lo siguiente:

```
funcion MTDF(root, f, d) retorna entero
        g := f;
        upperBound := +∞;
        lowerBound := -∞;
        mientras lowerBound < upperBound hacer
                si g = lowerBound entonces
                        β := g+1;
                sino
                        β := g;
                fsi
                g := AlphaBetaWithMemory(root, β-1, β, d);
                si g < β entonces
                        upperBound := g;
                sino
                        lowerBound := g;
                fsi
        fmientras
        retorna ( g )
ffuncion
```

5. Áreas Relacionadas

Los algoritmos de juegos como minimax, como bien se intuye por el nombre, están directamente relacionados con la "Teoría de Juegos", que a su vez está relacionada con la filosofía, la biología, las matemáticas, la lógica, la economía, los negocios, las ciencias políticas, las estrategias militares, e incluso programas televisivos como el estadounidense *Friend or foe?* o el programa catalán *Sis a Traició*.

La **teoría de juegos** es un área de la matemática aplicada que utiliza modelos para estudiar interacciones en estructuras formalizadas de incentivos (que son los juegos) y llevar a cabo **procesos de decisión**. Sus investigadores estudian las estrategias óptimas así como el comportamiento previsto y observado de individuos en juegos. Y afortunadamente puede representarse con árboles de juegos y estudiarse con algoritmos de búsqueda en juegos. Es más, algunos investigadores creen que encontrar el equilibrio de los juegos puede predecir cómo se comportarían las poblaciones humanas si se enfrentasen a situaciones análogas al juego estudiado.

Aunque tiene algunos puntos en común con la teoría de la decisión, la teoría de juegos estudia decisiones en entornos donde interaccionan individuos. En otras palabras, estudia la elección de la acción óptima cuando los costes y los beneficios de cada opción no están fijados de antemano, sino que dependen de las elecciones de otros individuos. Vamos, que indirectamente estamos tratando con individuos que compiten y que según sus movimientos logran un beneficio o una pérdida minimax.

En el siguiente apartado aparecen algunas aplicaciones de dichos algoritmos en campos de la teoría de juegos.

6. Aplicaciones

Desde hace varias décadas el humano ha estado intentando comprender su propia inteligencia intentando generar software capaz de razonar como un humano. Y qué hay más un humano que jugar a juegos e intentar ganarlos.

Por este motivo no es de extrañar que durante tanto tiempo se hayan estado generando algoritmos y máquinas específicas para jugar a juegos, sobretodo al ajedrez, donde los algoritmos minimax han sido explotados en infinidad de torneos, concursos, y competiciones varias.

Centrémonos por un momento en **el ajedrez**, considerado por mucho tiempo un juego de auténticos genios y estrategas.

Cuando el juego de ajedrez se inicia, las blancas eligen uno entre 20 movimientos posibles. Siguiendo el movimiento de las blancas, las negras tienen 20 opciones de respuesta, hagan los que hagan las blancas.

De acuerdo a ésto, 400 posiciones distintas pueden surgir sólo de la primera jugada de una partida de ajedrez. Luego de 2 jugadas el número de posiciones posibles crece sobre las 20.000. Y tras varias jugadas más la cifra será astronómica. Vamos, que el "árbol de ajedrez" posee más nodos que la cantidad de átomos presentes en la via láctea.

De acuerdo al reglamento del juego una partida es tablas (empate) si transcurren 50 jugadas sin que se haya realizado algún movimiento de peón y sin que se haya producido algún cambio de pieza, por lo cual según algunos cálculos una partida de ajedrez puede durar como máximo unas 3150 jugadas, por lo que el árbol de variantes puede tener una cantidad de posiciones limitada a esta cantidad.

Por lo tanto, si fuese posible examinar el árbol por completo, buscando todas las líneas de juego y sus conclusiones, podría determinarse cuál es el mejor movimiento inicial. Sin embargo, en la práctica el árbol es demasiado extenso para considerar este mecanismo, motivo por el que se han ido refinando los algoritmos de búsqueda previamente explicados en este documento con tal de lograr máquinas que jueguen mejor, pues no únicamente la CPU marcará su “inteligencia”. Ilustrémoslo con una breve visión histórica.

En los inicios de 1970 la búsqueda en árboles de variantes alcanzaba la cantidad aproximada de 200 posiciones por segundo. La computadora de propósito especial *Deep Thought* (Anantharaman, 1990) jugaba al ajedrez con un esquema de búsqueda con extensiones singulares, y era capaz de buscar hacia abajo hasta alrededor de 10 niveles. Mediante la heurística de extensiones singulares la computadora ibaa más allá, buscando combinaciones de mate en mitad del juego que incluyan hasta 37 movimientos. El evaluador estático consideraba el conteo de piezas, la colocación de éstas, la estructura de peones, los peones perdidos, y el ordenamiento de peones y torres en columnas. Más tarde la famosa máquina *Deep Blue* de IBM, que venció a Kasparov, ya era capaz de buscar

en 2.000.000 de posiciones por segundo. Hace cinco años los mejores programas ya lograban examinar todas las secuencias de movimientos con una profundidad de 8 a 10 movidas en el árbol (4 a 5 jugadas por bando). Y la mejora no se debe simplemente al *hardware* del jugador.

Además en otros juegos que no sean ajedrez también se han logrado **máquinas que sean campeonas mundiales**. En las damas chinas la máquina Chinook es campeona des de 1994. En Backgammon, el sistema TD-gammon es considerado uno de los tres mejores jugadores del mundo (1995). En Othello, Logistello (1994) es considerado muy superior a los humanos. Y en Go-Moku se tiene un algoritmo con el que el jugador que empieza siempre gana. Como véis, a la hora de jugar las máquinas nos van comiendo terreno.

Pero tras este inciso de culturilla histórica, remarcar que los algoritmos de juegos no solamente se utilizan en máquinas capaces de vencer al hombre, sinó que tienen muchísimas más aplicaciones más allá del ámbito lúdico. Los algoritmos presentes en este documento pueden aplicarse en todo el universo de la teoría de juegos.

Debido a ésto, los algoritmos de juegos pueden usarse en ámbitos tan dispares como la economía, la ética, o la biología entre otros.

En **economía y negocios** se han usado para analizar un amplio abanico de problemas económicos, incluyendo subastas, duopolios, oligopolios, la formación de redes sociales, y sistemas de votaciones. Estas investigaciones normalmente están enfocadas a conjuntos particulares de estrategias conocidos como "conceptos de solución". Estos conceptos de solución están basados normalmente en lo requerido por las normas de racionalidad perfecta. Las recompensas de los juegos normalmente representarán la utilidad de los jugadores individuales, y las recompensas representarán dinero, que se presume corresponden a la utilidad de un individuo. Esta presunción, sin embargo, puede no ser siempre correcta.

En **biología**, las recompensas de los juegos suelen interpretarse como la "adaptación". En este ámbito, los algoritmos (por la teoría de juegos) se emplean para entender muchos problemas diferentes. Se usaron por primera vez para explicar la evolución (y estabilidad) de las proporciones de sexos 1:1 (mismo número de machos que de hembras). Ronald Fisher sugirió en 1930 que la proporción 1:1 es el resultado de la acción de los individuos tratando de maximizar el número de sus nietos sujetos a la restricción de las fuerzas evolutivas.

Además, los biólogos han usado la teoría de juegos evolutiva y el concepto de estrategia evolutivamente estable para explicar el surgimiento de la comunicación animal (John Maynard Smith y Harper en el año 2003). El análisis algorítmico de juegos con señales y otros juegos de comunicación ha proporcionado nuevas interpretaciones acerca de la evolución de la comunicación en los animales.

Finalmente, los biólogos han usado el problema halcón-paloma (también conocido como problema de la gallina) para analizar la conducta combativa y la territorialidad.

En **ética** también existen algoritmos para tomar decisiones humanas, aunque no necesariamente deban ser de búsqueda. Por ejemplo, para el clásico dilema humano del prisionero, la mejor decisión determinista implementada (y que ganó el concurso donde se presentó) simplemente ocupaba 4 líneas en BASIC. Se llamaba *Tit for Tat* ("donde las dan, las toman"), fue idea de Anatol Rapoport, y se limitaba a cooperar la primera vez y a partir de ahí repetir siempre lo que decidió el compañero en la iteración anterior. Más tarde se le añadió una pequeña probabilidad de capacidad de perdón.

Finalmente añadir que algoritmos de búsqueda de juegos también se utilizaron en ciertas estrategias militares durante la Guerra Fría. Todos conocemos la predisposición de la tecnología en las guerras.

7. Bibliografía y Enlaces

Para desarrollar este documento se han utilizado las siguientes fuentes, donde por supuesto podrá encontrarse más información que la aquí expuesta:

- Apuntes y Transparencias de Resolución de Búsquedas, Javier Béjar, curso 2006/2007, UPC-FIB, consultadas durante el curso 2007/2008: http://www.lsi.upc.edu/~bejar/ia/teoria.html
- Apuntes Introducción a la Inteligencia Artificial de la UIB, consultados el 27/04/2008: http://dmi.uib.es/~abasolo/intart/2-juegos.html
- Chess Programming Wiki, consultada el 10/06/2008: http://chessprogramming.wikispaces.com/
- Apuntes sobre Búsqueda en Juegos del departamento DCCIA de la Universidad de Alicante, consultados el 10/06/2008: http://www.dccia.ua.es/dccia/inf/asignaturas/FIA/teor%EDa/tema2Juegos.pdf
- Apuntes sobre Juegos del departamento de Ciencias Computacionales del Instituto Nacional AOE de Mexico, consultados el 11/06/2008: http://ccc.inaoep.mx/~emorales/Cursos/Busqueda04/capitulo3.pdf
- Apuntes "Técnicas Avanzadas" del departamento de Automática y Computación de la Universidad Pública de Navarra: http://www.ayc.unavarra.es/miguel.pagola/tecnicas%20avanzadas.pdf
- NegaScout Wikipedia EN, consultado 12/06/2008: http://en.wikipedia.org/wiki/Negascout
- Teoría de Juegos Wikiedia ES, consultado el 14/06/2008: http://es.wikipedia.org/wiki/Teor%C3%ADa_de_juegos
- An introduction to the MTD(f) minimax algorithm, por Aske Plaat, consultado el 12/06/2008, donde puede consultarse mucha más información sobre el algoritmo MTD(f).

Además a continuación se presentan una serie de enlaces extra que también podrían resultar de interés para el lector:

- Website del profesor George Stockman (SSS*): http://www.cse.msu.edu/~stockman/
- Website de Judea Pearl (SCOUT): http://bayes.cs.ucla.edu/jp_home.html
- Notación Decimal de Dewey: http://en.wikipedia.org/wiki/Dewey_Decimal_Classification
- Información de IBM sobre Deep Blue: http://www.research.ibm.com/deepblue/
- Practica de estudiantes de la UAA de Paraguay donde comparan el algoritmo MiniMax con el MTD(f) en un agente que juega "Mastergoal": http://www.uaa.edu.py/jit_cita/estudiantes/ia001.pdf
- El famoso "Dilema del Prisionero" y sus variantes: http://es.wikipedia.org/wiki/Dilema_del_prisionero
- Website muy completo sobre la "Teoría de Juegos": http://www.gametheory.net/