Búsqueda en grafos

César Ignacio García Osorio
Área de Lenguajes y Sistemas Informáticos
Universidad de Burgos

Tipos de búsqueda

- Atendiendo a la información utilizada se distinguen dos tipos fundamentales de búsqueda:
  - Búsqueda sin información, exhaustiva o **ciega**: no utiliza ningún criterio para seleccionar una regla entre las aplicables.
  - Búsqueda con información o **heurística**: usan conocimiento sobre la naturaleza del problema para seleccionar reglas. Ej. Cola de cajera.

Búsqueda ciega

- La mejor forma de ilustrar la búsqueda ciega es usar un espacio de búsqueda con estructura de árbol.
- Hay un arco de un nodo n_1 a otro n_2 si el estado que representa n_2 se puede obtener aplicando un operador al estado que representa el nodo n_1. El nodo n_1 se llama **padre** y el nodo n_2 **hijo**.
- Se supone un árbol con un **factor de ramificación medio** igual a ***b***, y que el nodo que representa el estado final esta a una **profundidad *d*** (la profundidad del nodo raíz es 0).
- Tipos de búsqueda ciega:
  - Primero en profundidad.
  - Primero en anchura.
  - Primero en profundidad con "profundización iterativa".

Búsqueda primero en profundidad

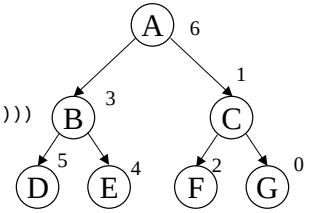
1. NODOS ← lista de nodos iniciales.
2. Si NODOS esta vacío, finalizar con fallo.
3. Si el primer elemento de NODOS es un nodo final, entonces finalizar con éxito.
4. En otro caso, hacer que NODOS sea la lista obtenida al concatenar los hijos del primer elemento de NODOS con el resto de elementos de NODOS.
5. Ir al paso 2.

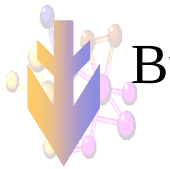
Búsqueda primero en profundidad

```
(defun dfs (nodes goalp next)
  (cond ((null nodes) nil)
        ;; Devuelve el primer nodo si es final
        ((funcall goalp (first nodes)) (first nodes))
        ;; Poner los hijos al frente de la lista
        (t (dfs (append (funcall next (first nodes))
                        (rest nodes))
                goalp
                next))))
```


Búsqueda primero en profundidad

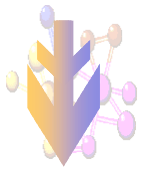
```
(defun make-TREE (label value children)
       (list 'tree label value children))
(defun TREE-label (tree) (second tree))
(defun TREE-value (tree) (third tree))
(defun TREE-children (tree) (fourth tree))
(defun TREE-print (tree) (princ (TREE-label tree)))

>(setq tree (make-TREE 'a 6
   (list (make-TREE 'b 3 (list (make-TREE 'd 5 nil)
                               (make-TREE 'e 4 nil)))
         (make-TREE 'c 1 (list (make-TREE 'f 2 nil)
                               (make-TREE 'g 0 nil))))))

(TREE A 6 ((TREE B 3 ((TREE D 5 NIL)
                      (TREE E 4 NIL)))
           (TREE C 1 ((TREE F 2 NIL)
                      (TREE G 0 NIL)))))
(dfs (list tree)
 #'(lambda (x) (TREE-print x) (eq 'g (TREE-label x)))
 #'TREE-children)
```





Búsqueda en profundidad con profundización iterativa

- Hace uso de una búsqueda en profundidad con profundidad limitada.
  1. NODOS ← lista de nodos iniciales.
  2. Si NODOS esta vacío, finalizar con fallo.
  3. Si el primer elemento de NODOS es un nodo final, entonces finalizar con éxito.
  4. Si la profundidad del primer elemento de NODOS es mayor que `max` eliminarlo de NODOS e ir al paso 2.
  5. En otro caso, hacer que NODOS sea la lista obtenida al concatenar los hijos del primer elemento de NODOS con el resto de elementos de NODOS.
  6. Ir al paso 2.



Búsqueda en profundidad con profundización iterativa

```
(defun dfs-fd (node goalp next depth max)
  (cond
     ;; Devuelve el primer nodo si es final
     ((funcall goalp node) node)
     ;; Salir si se ha llegado a la profundidad máxima
     ((= depth max) nil)
     ;; Incrementar la profundidad y continuar la búsqueda
     (t (some #'(lambda (n)
                  (dfs-fd n goalp next (+ 1 depth) max))
              (funcall next node)))))
```

Búsqueda en profundidad con profundización iterativa

```
(defun ids (start goalp next depth)
  ;; Primero busca a la profundidad actual.
  (or (dfs-fd start goalp next 0 depth)
  ;; Si es necesario busca a mayor  profundidad
     (ids start goalp next (+ 1 depth))))

>(ids tree
       #'(lambda (x) (TREE-print x)(eq 'g (TREE-
  label x)))
       #'TREE-children 0)
AABCABDECFG
(TREE G 0 NIL)
```


Notación para grafos

- **Grafo**: Un grafo ***G*** viene dado por un par *(N(G), A(G))*
  - *N(G)*: conjunto no vacío, posiblemente infinito, de elementos denominados **nodos**.
  - *A(G)*: familia de pares de elementos de *N(G)* denominados **arcos**.

    $a_{ij} = (n_i, n_j) = (n_j, n_i) = a_{ji}\ n_j, n_i \in N(G)$

    Si $\exists\ a_{ij}$ se dice que: n_i, n_j son **nodos adyacentes** y que n_i, n_j son **incidentes** al arco a_{ij}. *OJO*
- **Grafo dirigido**: Un grafo dirigido ***D*** es un par *(N(D), A(D))*
  - *N(D)*: conjunto no vacío, posiblemente infinito, de elementos denominados **nodos**.
  - *A(D)*: familia de pares *ordenados* de elementos de *N(D)* denominados **arcos**.

    $a_{ij} = n_i, n_j \neq a_{ji}\ n_j, n_i \in N(D)$

Notación para grafos

- **Camino**: Un camino ***C*** es una secuencia ordenada de nodos $C=[n_1, n_2, \ldots n_k]$ donde para cada n_i, $n_{i+1}\ \exists\ a_{i,\,i+1} \in A$. La **longitud de un camino** es el número total de nodos en el camino menos 1 (es decir, el número total de arcos). Se dice que existe una **conexión** entre dos nodos n_i, $n_j \in N$ si $\exists\ C=[n_i, \ldots n_j]$. Si todos los nodos de una grafo están *conectados*, se dice que es un **grafo conexo**.
- **Padres, hijos, hermanos, ascendientes y descendientes**: Entre los nodos de un grafo se pueden dar relaciones de "*parentesco*" dependiendo de cómo estén unidos mediante arcos.

Notación para grafos

- **Nodo raíz**: Es un nodo especial de los grafos dirigidos, $n_0 \in N(D)$ es un nodo raíz de $N(D)$ si n_0 no tiene padre y $\forall\ n_i \in N(D), \exists\ C=[n_0, \ldots n_i]$. Si un grafo dirigido tiene un nodo raíz se llama **grafo dirigido enraizado**. A los nodos que no tienen hijos se les llama **nodos terminales** o **nodos hoja**
- **Ciclo**: Un camino $C=[n_1, \ldots n_k]$ es un ciclo si $n_1=n_k$ y $n_i \neq n_j$ $j=2, 3, \ldots k-1$
- **Árbol**: Grafo conexo que no contiene ciclos.
- **Árbol dirigido**: Grafo dirigido enraizado en que cada nodo tiene un único padre (salvo el raíz).

Notación para grafos

- **Profundidad de un nodo**: En un grafo dirigido enraizado elentero que se asocia a cada nodo, definido recursivamente por:
  - profundidad nodo raíz, n_0: 0
  - profundidad de n_i: profundidad *nodo padre* + 1
- **Costo de un arco**:El costo c_{ij} de un arco a_{ij} es un número positivo asociado al arco. El **costo de un camino** $C=[n_{i1}, n_{i2}, ... n_{ik}]$ es la suma de los costos de los arcos que componen el camino.

Representación de grafos

- Representación explícita: cualquier representación que enumere los nodos y los arcos incidentes de cada nodo.
  - Ej Matriz de incidencia.
- Grafos dirigidos: Una forma conveniente (orientada al Lisp) de representación explícita es mediante una lista de *n* elementos (tanto como nodos):
  - $D= (l_1, l_2, ... l_n)$ con cada $l_i = (n_i, hijo_1, hijo_2, ... hijo_k)$ o de forma alternativa: $li = (n_i, padre_1, padre_2, ... padre_e)$.

Una de las ventajas en la representación en forma de lista es un menor consumo de memoria pues la matriz de incidencia suele ser dispersa.

Representación de grafos

- La representación explícita de un grafo es impracticable para grafos grandes e imposible para grafos infinitos.
- Se habla de representación implícita cuando se proporciona:
  - un nodo inicial.
  - un mecanismo que permita crear todos los sucesores de cualquier nodo.
- Las representaciones implícitas son interesantes si permiten hacer explícita cualquier parte de un grafo.

Representación de grafos

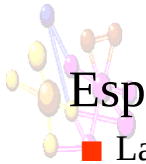
- En particular, nos van a interesar los grafos dirigidos enraizados localmente finitos.
  - **Grafo localmente finito**: cualquier nodo tiene un número finito de sucesores (el grafo puede ser infinito)
- Si la representación implícita de un grafo dirigido enraizado localmente finito proporciona como nodo inicial el nodo raíz, podemos hacer explícita cualquier parte del grafo.
- Paso de implícito a explícito:
  - **generar un nodo**: obtener un nodo a partir de su padre.
  - **explorar un nodo**: generar alguno de sus hijos.
  - **expandir un nodo**: generar todos sus hijos.

Repaso de conceptos

- Un **espacio de estados** define un conjunto de objetos entre los que estamos interesados en realizar la búsqueda.
- Un **estado** es una representación que describe la configuración de un problema.
- Los objetos están "relacionados" entre ellos a través de **operadores** que "transforman" un objeto en otro.

Repaso de conceptos

- Los operadores están formados por una **condición** de aplicación y una **acción** que transforma el estado.
- En el espacio de estados existen algunos **estados finales** (satisfacen algún criterio) a los que se quiere llegar a partir de los **estados iniciales** aplicando los operadores.
- En ocasiones es posible asociar al espacio una **métrica** que nos mide la distancia de los estados a un estado final.



Espacio de estados y grafos implícitos

- La representación de espacio de estados utiliza:
  - estados → configuración
  - operadores → pasos a la solución

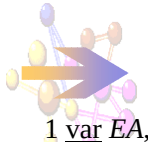
  Si representamos los estados por nodos y la aplicación de operadores por arcos queda definido de forma implícita un grafo dirigido *D*.
- Solución problema mediante grafos dirigidos
  - *NI*: conjunto de nodos etiquetados con estados de *I*.
  - *NF*: conjunto de nodos etiquetados con estados de *F*.
  - **solución**: hacer explícita una parte del grafo *D* que contenga un camino $C=[n_0, n_1, \ldots n_k]$ con $n_0 \in$ NI y $n_k \in NF$

Todas las estrategias de búsqueda en grafos que vamos a ver trabajan expandiendo nodos. Se diferencian por el criterio utilizado para seleccionar el nodo a expandir.

Implementación

- Vamos a a hacer explícito un árbol dirigido.
- Cada nodo n_i se representa por la lista: (n_i *padre-*n_i).
- En general, el espacio de estados define un grafo implícito, no un árbol pero los algoritmos sólo van a conservar un padre para cada nodo.
- Los algoritmos van a manejar dos listas:
  - ABIERTOS: nodos generados que hay que expandir.
  - CERRADOS: nodos ya expandidos

  CERRADOS contiene los nodos que de momento no se van a volver a expandir y ABIERTOS la **frontera de exploración**. Entre abiertos y cerrados se obtiene una representación explícita de un árbol incluido en el grafo.
- El tipo de algoritmo que se obtiene depende de la gestión de los elementos de ABIERTOS.



Primero en achura

1 var *EA*, *SUC*
2 *ABIERTOS* ← (n_0)
3 *CERRADOS* ← ( )
4 while *ABIERTOS* ≠ ( ) do
begin
4.1 *EA* ← primer elemento de *ABIERTOS*
4.2 if *EA* es una meta then return camino [n_0, n_1, ... *EA*]
4.3 eliminar el primer elemento de *ABIERTOS*
4.4 expandir *EA*, colocando sus hijos en *SUC*
4.5 colocar *EA* en *CERRADOS*
4.6 eliminar de *SUC* cualquier elemento que estuviese en *ABIERTOS* o en *CERRADOS*
4.7 colocar los restantes elementos de SUC **al final** de *ABIERTOS*, como hijos de *EA*.
end
5 return "fallo"

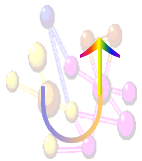
Elección del tipo de búsqueda

- Algunos criterios orientativos que permiten elegir entre búsqueda en anchura y en profundidad son:
  - factor de ramificación medio *b*:
    - si es grande se prefiere la búsqueda en profundidad
    - si es pequeño la búsqueda en anchura
  - longitud de camino hasta la meta *d*:
    - si es grande se prefiere la búsqueda en profundidad
    - si es pequeño la búsqueda en anchura
  - número de estados finales:
    - si hay muchos se prefiere en profundidad
    - si son pocos en anchura
  - necesidad del camino más corto:
    - si es necesario el camino más corto se utiliza la búsqueda en anchura o la profundización iterativa.
  - finitud del espacio de estados:
    - si el EE es infinito o si la profundidad es mucho mayor que *d* se prefiere la búsqueda en anchura o la profundización iterativa.

Primero en profundidad mejorado

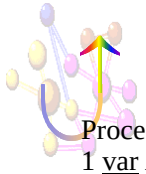
1 var *EA*, *SUC*
2 *ABIERTOS* ← (n_0)
3 *CERRADOS* ← ( )
4 while *ABIERTOS* ≠ ( ) do
begin
4.1 *EA* ← primer elemento de *ABIERTOS*
4.2 if *EA* es una meta then return camino [n_0, n_1, ... *EA*]
4.3 eliminar el primer elemento de *ABIERTOS*
4.4 if profundidad EA > LIMITE then limpiar CERRADOS else
begin
4.4.1 colocar EA al principio de CERRADOS
4.4.2 expandir EA, colocando sus hijos en SUC
4.4.3 eliminar de SUC cualquier elemento que estuviese en ABIERTOS o en CERRADOS
4.4.4 if SUC = ( ) then limpiar CERRADOS else colocar los elementos de SUC al principio de ABIERTOS como hijos de EA
end
end
5 return ``fallo"

La acción de ***limpiar*** CERRADOS consiste en ir eliminando los nodos de CERRADOS hasta encontrar uno con sucesores en ABIERTOS.



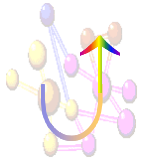
Búsqueda retroactiva (backtracking)

- Búsqueda tentativa más sencilla de implementar, basta con mantener la secuencia de estados recorridos.
- Comienza por el estado inicial y aplica reglas hasta que alcanza una meta o punto muerto, manteniendo la secuencia de estados recorridos (camino).
- Si alcanza la meta devuelve el camino o la secuencia de reglas.
- Si alcanza un punto muerto, vuelve al estado inmediatamente anterior, descarta el actual y continua buscando.



Búsqueda retroactiva (2)

```
Procedimiento RETROCESO(EA)
1 var NEA, R, REGLAS, SOL
2 if TERM(E) then return()
3 if SINSALIDA(EA) then return fallo
4 REGLAS ← APLIREGLAS(EA)
5 while REGLAS no esta vacío do
    begin
    5.1 R ← primer elemento de REGLAS
    5.2 REGLAS ← REGLAS - R
    5.3 NEA ← NSTAB(EA,R)
    5.4 SOL ← RETROCESO(NEA)
    5.5 if SOL ≠ fallo then begin
          5.5.1 SOL ← añadir R al principio de SOL
          5.5.2 return SOL
          end
     end
6 return fallo
```

Búsqueda retroactiva (3)

- Posibles mejoras:
  - evitar ciclos
  - limitar longitud de camino
  - búsqueda informada (búsqueda retroactiva informada)
- Primero en profundidad con sólo el camino vs. Retroactiva
  - el orden de exploración es el mismo
  - la única diferencia es que retroactiva genera un hijo cada vez y profundidad lo hace simultáneamente
  - en profundidad se evitan ciclo
  - debido a su menor complejidad espacial se suele preferir el algoritmo retroactivo, especialmente si el grafo se reduce a un árbol

Búsquedas heurísticas

- Todas las estrategias de búsqueda exhaustiva tienen una complejidad temporal exponencial $O(b^d)$.
- La única forma de intentar reducir esta complejidad es emplear información sobre la estructura del espacio de búsqueda.
- Se utiliza una función de evaluación que toma valores para cada nodo del grafo, estimando su distancia a un nodo objetivo, permitiendo seleccionar el siguiente nodo a explorar.
- Una heurística es una regla o método que guía la decisión que hacemos al elegir un nodo que explorar, aunque no siempre permite hacer la mejor elección.

Tipos de búsquedas heurísticas

- Métodos exactos
  - Primero el mejor (Best First)
  - Beam Search (Búsqueda en haz)
  - Algoritmo *A*
  - Algoritmo A^*
- Métodos aproximados
  - Escalada simple (simple hill climbing)
  - Escalada profunda (steepest-ascent hill climbing)
- Búsquedas con contrincante
  - Búsqueda Mini-Max
  - Búsqueda Alfa-Beta

Búsqueda primero el mejor

- Combinación de amplitud y profundidad.
- Seguir un camino, pero pasar a otro cuando sea más prometedor (según la función de evaluación *f*)
- Se trata de realizar:
  - Búsqueda en profundidad de la rama más prometedora.
  - A medida que se avanza en la rama sin solución, la más prometedora pasa a ser otra de nivel superior (amplitud)
  - Se guardan las hojas no exploradas hasta el momento.
- Si la búsqueda es en un grafo, hay que evitar que se produzcan ciclos.
- La estrategia primero el mejor se caracteriza por:
  - expandir primero el nodo con un menor valor de *f*
  - si dos caminos llevan al mismo nodo (en grafos), sólo se conserva el camino que proporciona un menor valor de *f*.

Primero el mejor en un árbol

1. *NODOS* ← lista ordenada de nodos iniciales.
2. Si *NODOS* esta vacío, finalizar con fallo.
3. *N* ← primer elemento de *NODOS*, y eliminar *N* de *NODOS*
4. Si *N* es un nodo final, entonces finalizar con éxito.
5. En otro caso, añadir de forma ordenada los hijos de *N* a *NODOS*
6. Ir al paso 2.

```
(defun best (nodes goalp next comparep)
  (cond ((null nodes) nil)
        ;; Devuelve el primer nodo si es final
        ((funcall goalp (first nodes)) (first nodes))
        ;; Poner los hijos al frente de la lista y despues
           ordenar
        (t (best (sort (append (funcall next (first nodes))
                               (rest nodes)) comparep)
                 goalp next comparep))))
```


Primero el mejor en un grafo

- El algoritmo utiliza dos listas: *ABIERTOS* y *CERRADOS*.
- A diferencia de los algoritmos anteriores, este algoritmo puede llevar un nodo de *CERRADOS* a *ABIERTOS*
- El algoritmo hace explícito un árbol, pero para cada nodo mantendrá como padre aquel que se encuentre en un camino que proporcione un menor valor de *f*.
- Para ello, cada vez que genera un nodo repetido tiene que:
  - calcular nuevo valor de *f*
  - si el valor de *f* es menor que el anterior, modificar el padre y anotar nuevo valor de *f*
  - si el nodo estaba en *CERRADOS*, llevarlo a *ABIERTOS*
- También hay que modificar la representación de un nodo: $(n_i$ *padre* $f(n_i))$. Donde $f(n_i)$ es el menor valor de *f* encontrado por el algoritmo para n_i y padre permitido obtener este valor de *f*.

Primero el mejor en grafos

1. *ABIERTOS* ← (n_0); *CERRADOS* ← ( )
2. si *ABIERTOS* es la lista vacía, fin con fallo.
3. *EA* ← primer elemento de *ABIERTOS*. Eliminar *EA* de *ABIERTOS* y llevarlo a *CERRADOS*.
4. Expandir nodo *EA*, generando todos sus sucesores como hijos de *EA*.
5. Si alguno de los sucesores de *EA* es una meta, fin con éxito. Devolver el camino hasta la meta.
6. Para cada sucesor *q* de *EA*:
   a) Calcular *f(q)*
   b) Si q no estaba en *ABIERTOS* ni en *CERRADOS*, colocarlo en *ABIERTOS*, asignando el valor *f(q)*
   c) Si q estaba en *ABIERTOS* o en *CERRADOS*, comparar el nuevo valor *f(q)* con el anterior. Si el anterior es menor o igual, descartar el nodo recién generado. Si el nuevo es menor, colocar EA como nuevo padre y asignar el nuevo valor *f(q)*. Si estaba en *CERRADOS* eliminarle de *CERRADOS* y llevarle a *ABIERTOS*.
7. Reordenar *ABIERTOS* según valores crecientes de *f*.
8. Ir a 2.

Algoritmo *A*

- Caso particular de "primero el mejor", utilizando una función de evaluación

  $f^*(n)$=costo del camino solución óptimo que pasa por n
  - puede no estar definida para todo n
  - aunque este definida, no podemos calcular su valor
  - la única forma de calcular f^* es conociendo los caminos solución óptimos para cada nodo, con lo cual tendríamos resuelto el problema.
- Podemos descomponer $f^*(n)=g^*(n)+h^*(n)$ donde:
  - $g^*(n)$: costo del camino mínimo de n_0 a n.
  - $h^*(n)$: costo del camino mínimo de n a un nodo meta.
- Seguimos sin conocer f^*, pero nos proporciona un método operativo para estimarla.

Algoritmo *A*

- Estimaremos f^* mediante una función $f(n)=g(n)+h(n)$, donde:
  - $g(n)$: costo del camino mínimo de n_0 a n encontrado hasta el momento por el algoritmo de exploración, por tanto $g(n)\geq g^*(n)$
  - $h(n)$: estimación del costo del camino mínimo de n hasta una meta cualquiera.
- De todos los caminos explorados el algoritmo ha de mantener el de menor coste encontrado.
- La heurística del problema está en la función h
- Si utilizamos f para seleccionar los nodos de ABIERTOS $\Rightarrow$Algorítmo *A*
- Si $\forall n\ h(n)=0$ y $g(n)=profund(n) \Rightarrow$ búsqueda en anchura
- **Función minorante**: h es minorante de h^* sii $\forall n\ h(n)\leq h^*(n)$

Algoritmo *A**

- Un algoritmo es **admisible** sii encuentra una solución óptima (mínimo costo) siempre que exista solución.
- A un algoritmo *A* admisible se le denomina *A**.
- El algoritmo *A* es admisible si la función h es minorante de h^* (es decir $\forall n\ h(n)\leq h^*(n)$)

Algoritmo A*

1. *ABIERTOS* $\leftarrow (n_0)$; *CERRADOS* $\leftarrow$ ( )
2. Si *ABIERTOS* es la lista vacía, fin con fallo.
3. *EA* $\leftarrow$ primer elemento de *ABIERTOS*. Eliminar *EA* de *ABIERTOS* y llevarlo a *CERRADOS*.
4. Si *EA* es una meta, fin con éxito. Devolver el camino hasta la meta.
5. Expandir nodo *EA*, generando todos sus sucesores como hijos de *EA*.
6. Para cada sucesor q de *EA*:
   a) Calcular $g(q)=g(EA)+c(EA,q)$
   b) Si q no estaba en *ABIERTOS* ni en *CERRADOS*, calcular $f(q)=g(q)+h(q)$, añadir q a *ABIERTOS* como hijo de *EA*, asignando los valores $f(q)$ y $g(q)$.
   c) Si q estaba en *ABIERTOS* o en *CERRADOS*, comparar el nuevo valor $g(q)$ con el anterior. Si el anterior es menor o igual, descartar el nodo recién generado. Si el nuevo es menor, colocar *EA* como nuevo padre y asignar los nuevos valores $g(q)$ y $f(q)$.
   d) Si se ha modificado el padre de q y estaba en *CERRADOS*, eliminarle de *CERRADOS* y llevarle a *ABIERTOS*.
7. Reordenar *ABIERTOS* según valores crecientes de f.
8. Ir a 2.

Propiedades formales del A*

Completo: Un algoritmo de búsqueda es completo sii termina con una solución siempre que esta exista.

Admisible: Un algoritmo de búsqueda es admisible sii encuentra una solución óptima (mínimo costo) siempre que exista solución

- **Prop1**: El algoritmo primero el mejor termina para grafos finitos
- **Prop2**: Al comienzo de cada iteración, el algoritmo primero el mejor tiene en *ABIERTOS* un nodo que esta en un camino solución, siempre que esta exista.
- **Prop3**: El algoritmo primero el mejor es completo para grafos finitos
- **Teo1**: *A** es completo incluso en grafos infinitos (localmente finitos).
- **Prop4**: En todo momento antes de que *A** termine, existe un nodo $n \in C^*_{n_0,\gamma}$ en *ABIERTOS* con $f(n) \leq C^*$ ($C^*_{n_0,\gamma}$≡camino solución, C^*≡coste $C^*_{n_0,\gamma}$)
- **Teo2**: *A** es admisible en grafos localmente finitos

Eficiencia de A*

- Se puede demostrar que A* es el algoritmo más eficiente, en el sentido de expandir el menor número de nodos garantizando la solución óptima.
- La admisibilidad de A* se obtiene a cambio de un alto costo de computación: núm. nodos generados + cálculo *h*:
  - si $h << h^*$, A* realiza una búsqueda similar a primero en anchura, generando gran núm. De nodos.
  - Si $h \approx h^*$, el cálculo de h tiene un costo computacional elevado.
- Para intentar mejorar la eficiencia de A* se ha propuesto considerar por separado los efecto de *g* y *h*, usando la función:
  - $f_\omega(n)=(1-\omega)g(n)+\omega h(n)$, $\omega \in [0,1]$
    - $\omega=0$ costo uniforme (anchura si costo unitario)
    - $\omega=1/2$ A* si $h \leq h^*$
    - $\omega=1$ primero el mejor (no tengo en cuenta el costo del camino)

Escalada simple

1. Evaluar el estado inicial
2. Si estado meta, final con éxito. Sino, hacer estado actual = estado inicial.
3. Mientras no se encuentre solución y haya operadores nuevos a aplicar al estado actual:
   (a) Seleccionar un operador que no se haya aplicado al estado actual y producir un nuevo estado.
   (b) Evaluar el nuevo estado
      i. Si es meta, final con éxito
      ii. Si no lo es, pero es mejor que el actual, hacer que sea el actual
      iii. Ir al paso 3

Escalada profunda

1. Evaluar el estado inicial
2. Si estado meta, final con éxito. Sino, hacer estado actual = estado inicial.
3. Mientras no se encuentre solución:
   (a) Crear *SUC* como lista de todos los posibles sucesores
   (b) Evaluar todos los elementos de *SUC*
      i. Si alguno de ellos es meta, final con éxito
      ii. Si no hacer que el estado actual sea el mejor elemento de *SUC*, siempre que sea mejor que él.
      iii. Ir al paso 3

Problemas Escalada

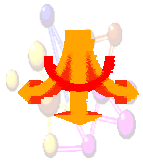
- Máximos locales
  - Estado en el que todos los descendientes son peores que él, y además no es nodo meta
- Altiplanicies (mesetas)
  - Todos los estados en un determinado vecindario tienen el mismo valor de evaluación.
  - No es posible determinar la mejor dirección de movimiento mediante comparaciones locales

Soluciones escalada

- Volver a un estado reciente del pasado e intentar en otra dirección (Backtracking)
- Realizar un salto grande en una dirección arbitraria para intentar ir a para a otra región del espacio de búsqueda
- Moverse cada vez en varias direcciones al mismo tiempo (no solo en la mejor)
- Otros algoritmos: enfriamiento simulado, algoritmos genéticos.

Diferencias entre Primero el Mejor y Escalada

- Para cada nodo actual se selecciona un único nodo de entre sus descendientes
- El resto de los descendientes no seleccionados se almacenan
- Los descendientes del nodo seleccionado compiten con el resto de los nodos almacenados para ser el nuevo nodo actual
- Siempre se selecciona un nuevo nodo como nodo actual, aunque no sea el mejor que el nodo actual que los precede.



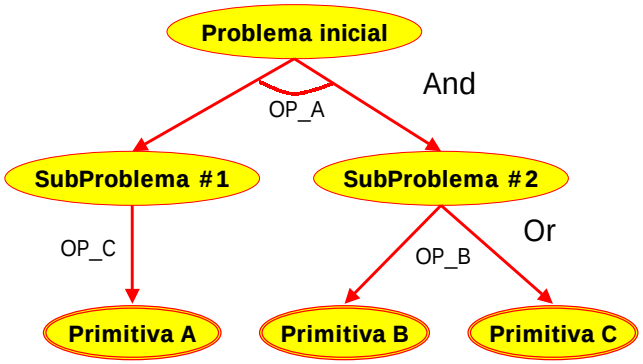
Problemas descomponibles (1)

- Representación de ***espacio de estados***: transformación de un problema es otro mediante la aplicación de una regla, con la esperanza de que el nuevo problema sea más fácil de resolver.
- Otros problemas responden, de forma natural, a un modelo de solución distinto, denominado ***descomposición (reducción) de problemas***:
  - Integración simbólica.
  - Demostración automática de teoremas.

Problemas descomponibles (2)

- Dado un problema (estado), descomponerlo en subproblemas (estados) más simples.
- La solución del problema exige la solución independiente de cada uno de los subproblemas.
- El método de descomposición también usará estados y operadores, sólo que ahora:
  - ***estado*** → descripción problema.
  - ***operador*** → transforma un problema en uno o más subproblemas.
- El estado inicial será el problema a resolver y los estados finales son problemas que sabemos resolver (***primitivas***).

Problemas descomponibles (3)



Problemas descomponibles (4)

Un ejemplo:

Sean p,q,r,s,t,y u proposiciones lógicas relacionadas según:

$r \wedge s \rightarrow p$

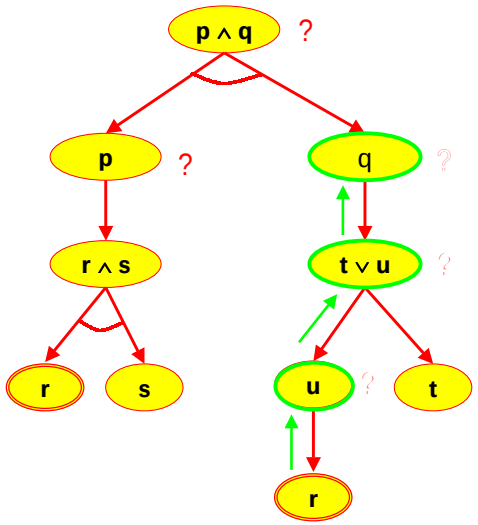
$t \vee u \rightarrow q$

$r \rightarrow u$

r

El problema:
demostrar el teorema

$p \wedge q$



Grafos Y/O

- Un grafo Y/O es un hipergrafo formado por hiperarcos:
  - Cada nodo tiene un conjunto de hiperarcos que conecta a un nodo con otros
  - Cada hiperarco conecta a un nodo con un conjunto de sucesores
  - Un hiperarco es *k-conector* si conecta un nodo con *k* sucesores

Grafos Y/O

- Un camino *C*, en un hipergrafo es una secuencia ordenada $C=[n_1, n_2, ..., n_k]$, donde para cada par de nodos consecutivos $n_i, n_{i+1} \in C$, n_{i+1} es sucesor de n_i a través de algún *k*-conector.
- *G'* es un grafo solución de *n* a *NF* sii se cumple una de las siguientes soluciones
  1) $n \in NF$ y *G'* se reduce a *n*
  2) $n \notin NF$ y $\exists$ *k*-conector $[n_1, n_2, ..., n_k]$ y *k* subgrafos solución, G_i' desde cada n_i a *NF*. En este caso, *G'* esta formado por el nodo *n*, el *k*-conector $[n, n_1, ...n_k]$ y cada uno de los subgrafos G_i'

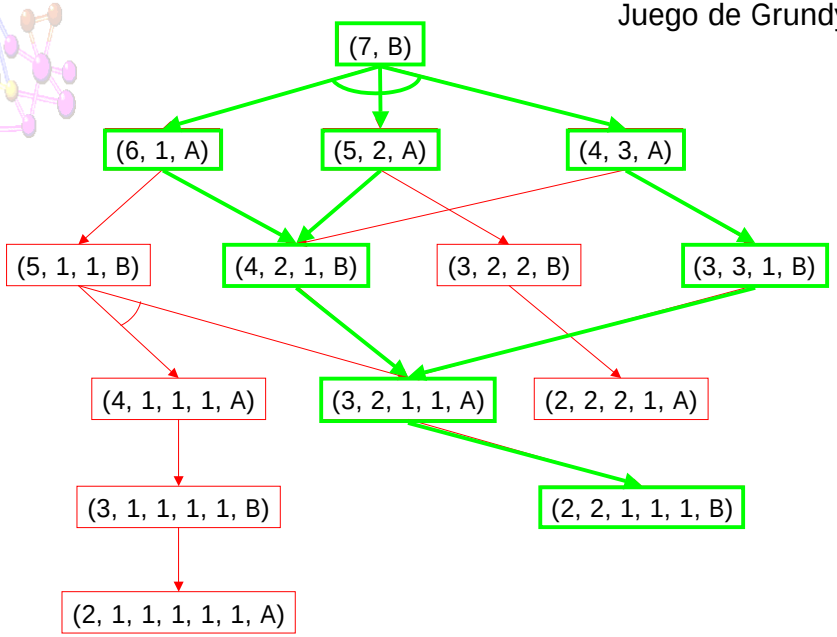
Búsqueda con contrincante (1)

- Problema: ganar un juego contra un oponente.
- El juego implica actuación alterna de dos jugadores.
- Juego de suma cero: la victoria de uno supone la derrota del otro.
- Se conoce la posibilidad de movimientos de cada jugador.
- Se conoce el movimiento del contrincante.
- Se desconoce la estrategia del contrincante.
- No interviene el azar.
- Se pueden especificar las situaciones de victoria, derrota, empate.

Búsqueda con contrincante (2)

- Grafo de juego: representación explícita de todos los posibles movimientos de un juego:
  - se puede interpretar como un grafo Y/O.
  - cada subgrafo solución del nodo inicial a un nodo terminal que representa una posición ganadora es una estrategia victoriosa.

Juego de Grundy



Búsqueda exhaustiva

- Esta búsqueda exhaustiva de un grafo solución es la ÚNICA que garantiza la victoria, pero es impracticable en la mayoría de los juegos.
  - Damas: 10^{40} nodos
    - (10^{21} siglos para generar el árbol, suponiendo que cada sucesor pudiera generarse en un tercio de nanosegundo)
  - Ajedrez: 10^{120} nodos

Alternativas a la búsqueda exhaustiva (1)

- Uso función evaluación estática, *f*

  Utilizar una función heurística que nos mida la calidad de una posición dada.

  Obtención movimiento:

  1) generar sucesores de una posición.
  2) calcular *f* para cada sucesor.
  3) seleccionar el movimiento que proporcione un mejor valor de *f*.

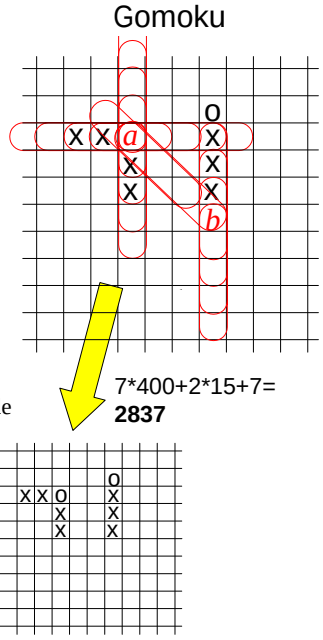
El objetivo del juego es conseguir cinco marcas en una línea (horizontal, vertical o diagonal). Nosotros jugamos con O y pretendemos conseguir la línea al tiempo que evitamos que lo haga nuestro oponente.

En vez de considerar el tablero como un conjunto de escajes se puede ver como un espacio de 5-tuplas entrelazadas.

Dependiendo del contenido de cada 5-tupla se le da una puntuación distinta, la mayor puntuación será la de la tupla "OOOO" ya que jugar en ella nos da la victoria, seguida de la tupla "XXXX" ya que jugando en ella evitamos la victoria del oponente.La puntuación mínima para tuplas como "XXOX".

La puntuación de una casilla/sucesor vendrá dada por la suma de todas las tuplas a las que pertenece.

- Tupla vacía: 7
- Tupla con una X: 15
- Tupla con una O: 35
- Tupla con dos Xs: 400
- Tupla con dos Os: 800
- Tupla con tres Xs:1800
- Tupla con tres Os: 15000
- Tupla con cuatro Xs: 100000
- Tupla con cuatro Os: 800000



Inconveniente:

- si la función *f* no es precisa, el nivel de juego es muy bajo.
- diseñar una buena función no es una tarea sencilla (ej. ajedrez)
- incluso funciones *f* sofisticadas proporcionan malas estimaciones de la calidad de la posición.

Alternativas a la búsqueda exhaustiva (2)

- Función de evaluación estática + exploración limitada:
  1) generar un parte del grafo de juego.
  2) calcular *f* en la frontera de exploración.
  3) usar esos valores para estimar la calidad de sus antecesores, hasta los hijos de la posición actual.
  4) elegir el movimiento que lleve al hijo de mejor calidad.
  - Suposición subyacente: la calidad de una posición se clarifica a medida que exploramos sus sucesores, lo que nos permite alcanzar un nivel de juego razonable incluso con funciones poco precisas sin más que aumentar el límite de exploración.

Regla miniMAX

- Llamando $V(J)$ al valor que asignamos a un nodo, la regla minimax es:
  1) Si J esta en la frontera de exploración, $V(J)=f(J)$.
  2) Si J es un nodo *MAX*, $V(J)$ es el máximo valor de sus hijos.
  3) Si J es un nodo *MIN*, $V(J)$ es el mínimo valor de sus hijos.
- Exploración limitada+función de evaluación heurística (f).
- f se define desde el punto de vista del ordenador (*MAX*):
  Posición ganadora: valor máximo.
  Posición perdedora: valor mínimo.

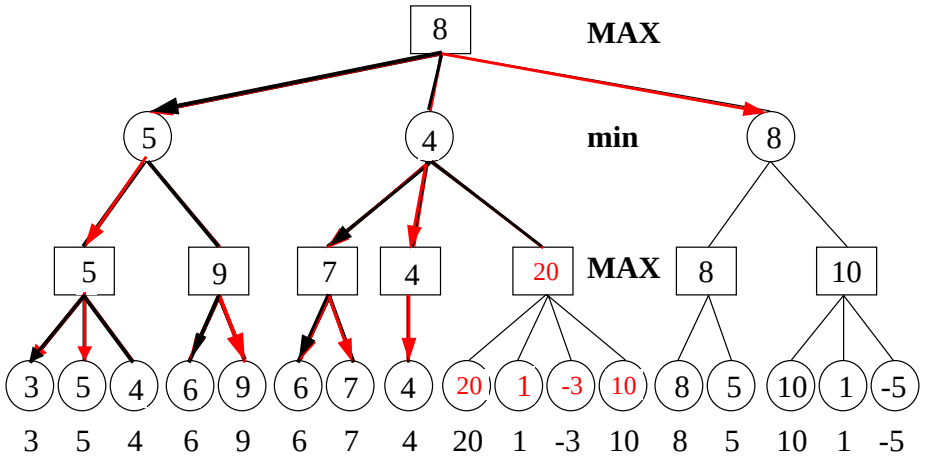
Implementación miniMAX

MINIMAX(nodo-actual, 0)

Procedimiento **MINIMAX(***NODO*, *PROF***)**

1. Si *PROF* ≥ *LIMITE* devolver (*nil*, *f*(*NODO*))
2. Si *POS-GAN*(*NODO*), *POS-PER*(*NODO*), *POS-TAB*(*NODO*) devolver *MAX-VAL*, *MIN-VAL*, 0 respectivamente.
3. Mientras no se hayan explorado todos los hijos de *NODO*
   3.1 genera J_k, *k*-esimo sucesor de NODO
   3.2 evaluar $V(J_k)$ (llamando recursivamente a MINIMAX(J_k, *PROF*+1)
   3.3 si *k*=1, hacer $VA \leftarrow V(J_k)$ y $SUC \leftarrow J_1$. Si $k \geq 2$
   a) Si *NODO* es *MAX* y $V(J_k)>VA$, hacer $CV \leftarrow V(J_k)$ y $SUC \leftarrow J_k$
   b) Si *NODO* es *MIN* y $V(J_k)<VA$, hacer $CV \leftarrow V(J_k)$ y $SUC \leftarrow J_k$
4. Devolver (*SUC*, *VA*)

Ejemplo miniMAX



Alfa-Beta

- Se van a usar dos nuevas variables: *ALFA* y *BETA*
  - nodos *MAX*: *ALFA* cota inferior del valor del nodo.
    - obtención *ALFA*: mayor, hasta el momento, de los valores finales llevados hacia atrás de sus sucesores.
  - nodos *MIN*: *BETA* cota superior del valor del nodo.
    - obtención *BETA*: menor, hasta el momento, de los valores finales llevados hacia atrás de sus sucesores.
- Cortes *ALFA*: Si un nodo *MAX* tiene un valor *ALFA* ≥ *BETA* de cualquier nodo *MIN* previo, se suspende la exploración para este nodo, asignándole como valor su valor *ALFA*.
- Cortes *BETA*: Si un nodo *MIN* tiene un valor *BETA* ≤ *ALFA* de cualquier nodo *MAX* previo, se suspende la exploración para este nodo, asignándole como valor su valor *BETA*.

Principio alfa-beta: Si tiene una idea que indudablemente es mala, no pierda el tiempo comprobando qué tan mala es

Implementación alfa-beta

ALFA-BETA(nodo-actual, – ∞, +∞, 0)

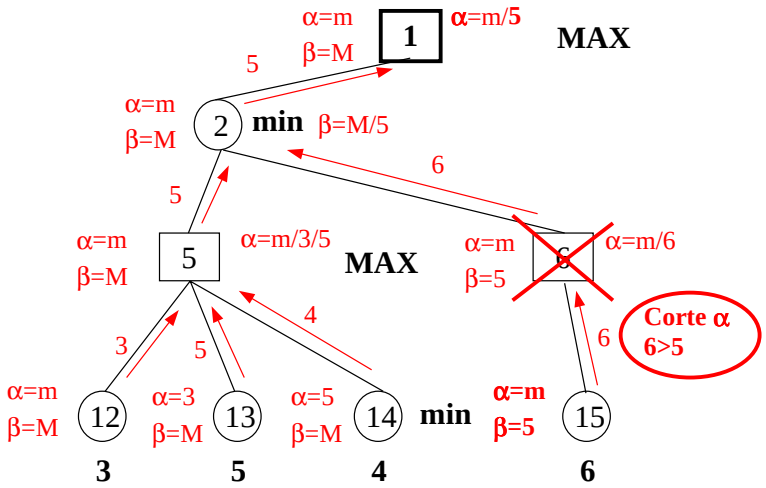
Procedimiento **ALFA-BETA(***NODO*, *ALFA*, *BETA*, *PROF***)**

1. Si *PROF* ≥ *LIMITE* devolver *f*(*NODO*)
2. Si *POS-GAN*(*NODO*), *POS-PER*(*NODO*), *POS-TAB*(*NODO*) devolver *MAX-VAL*, *MIN-VAL*, 0 respectivamente.
3. Si *NODO* es MAX
   - 3.1.a) Para todo *Si* sucesor de *NODO* y *ALFA*<*BETA*
     AL-BE= *ALFA-BETA*(*Si*, *ALFA*, *BETA*, *PROF*+1)
     ALFA=*MAX*(*ALFA*, *AL-BE*)
   - 3.1.b) Devolver *ALFA*

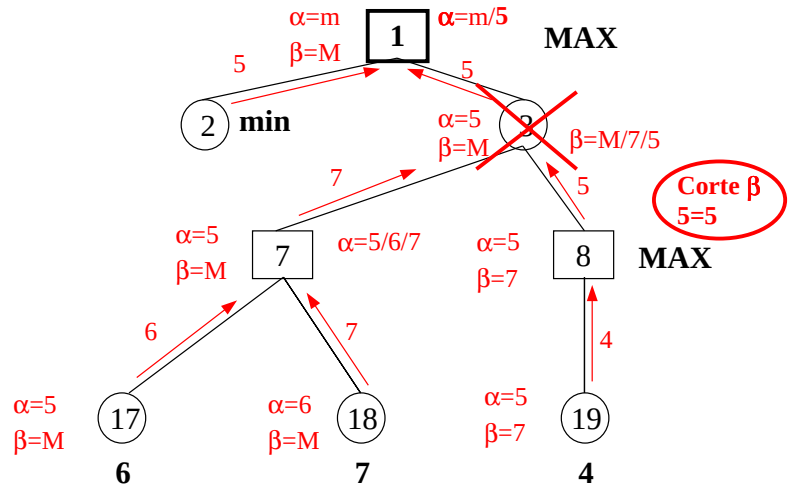
Si No

   - 3.2.a) Para todo *Si* sucesor de *NODO* y *BETA*>*ALFA*
     AL-BE= *ALFA-BETA*(*Si*, *ALFA*, *BETA*, *PROF*+1)
     BETA=*MIN*(*BETA*, *AL-BE*)
   - 3.2.b) Devolver *BETA*

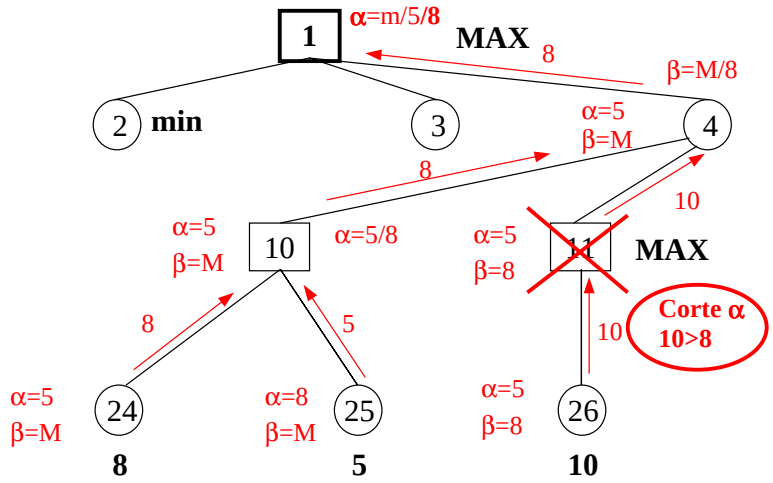
Ejemplo alfa-beta (1)



Ejemplo alfa-beta (2)

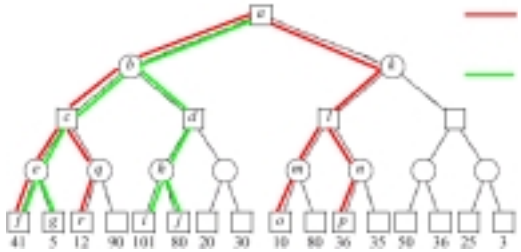


Ejemplo alfa-beta (3)



Mejora conseguida

- Para un árbol con factor de ramificación medio b y profundidad p, se puede llegar a calcular el valor miniMAX del nodo raíz generando únicamente $b^{\lfloor p/2 \rfloor} + b^{\lceil p/2 \rceil} - 1$ nodos hoja (si p es par $2b^{p/2}-1$, si p impar $b^{(p+1)/2}+b^{(p-1)/2}-1$ $2b^{p/2}-1$).
- Ese óptimo se conseguiría explorando en los nodos MAX primero los sucesores de mayor valor y en los nodos min primero los de menor valor.



Estrategia para el jugado min

Estrategia para el jugado MAX

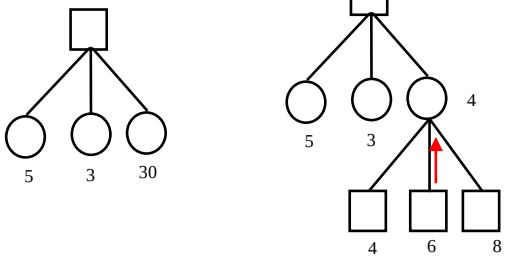
El min busca una cota superior para mis ganancias yo busco una cota inferior. El valor miniMAX será la menor de las cotas superiores que encuentra min y la mayor de las cotas inferiores que yo encuentre. El mínimo número de nodos necesarios para determinar dicho valor debe incluir al menos las estrategias ganadoras para el min y el MAX

Efecto horizonte

- Esta provocado por la profundidad fija. No “se ve” que es lo que ocurre más allá.
- Al fijar la profundidad de la búsqueda puede ocurrir que una situación sea muy ventajosa para un jugador, pero que si examináramos en el siguiente nivel se puede compensar y hasta ser desventajosa. El procedimiento se decidiría por un movimiento de forma engañosa
- Ejemplo: en ajedrez cuando sacrificamos piezas para luego capturar piezas más importantes, peón a punto coronar.
- Solución: búsqueda secundaria o de profundidad variable.

Equilibrio

- Problema relacionado con el anterior, puesto que el efecto horizonte provoca falta de equilibrio.
- Ejemplo:



- Solución: también la búsqueda secundaria o de profundizar variable.

Búsqueda secundaria

- En este tipo de búsqueda, no se estudia el árbol hasta una profundidad fija sino que se varía la profundidad de búsqueda.
- A los nodos en los que se estudia a más profundidad que a la profundidad máxima se les denomina "**extensiones selectivas**" y representan casos forzados.
- La forma de saber cuándo una situación es un caso forzado suele ser dependiente del dominio.
- Ejemplo: en el ajedrez se puede estudiar por debajo de aquello nodos en los que se capture una pieza.
- Otra solución consiste en utilizar información independiente del dominio como son las "**extensiones singulares**"
- Una jugada es singular si devuelve un valor mucho mayor que sus jugadas hermanas (con el mismo padre). Es decir, si $n \in suc(n')$ y $f(n) >> \max\{ f(n'')\}\ \forall n'' \in \{suc(n')-n\}$ se dice que n es un nodo singular.
- En estos nodos es donde se realizará una búsqueda secundaria.

Otras mejoras del alfa-beta (1)

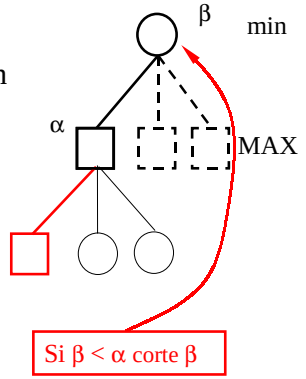
- **Ordenación nodos en el Alfa-Beta** (Slagel y Dixon)
- Se ordenan los sucesores de una posición por su valor estático, para producir el mayor número de cortes α y β.
- Si los sucesores de mayor valor, en una posición "MAX", y los peores sucesores, en una posición "MIN", se ponen delante de los demás, se conseguirán cuanto antes todas las podas posibles.
- Las podas se conseguirán cuanto antes se alcancen valores altos/bajos para los α/β.
- Para ordenar los sucesores se puede utilizar una función de evaluación más sencilla que la que se utiliza normalmente en los nodos hojas.
- El número de hojas de un árbol de profundidad *p* generado por una búsqueda óptima Alfa-Beta es aproximadamente igual al número de hojas que se habrían generado, sin el método Alfa-Beta, a la profundidad *p*/2.
- Para la misma capacidad de almacenamiento, el método Alfa-Beta con sucesores en el orden perfecto permitiría doblar la profundidad de búsqueda.
- El orden perfecto es imposible de lograr, se debe buscar la mejor aproximación con miras a reducir el alto coste.

Otras mejoras del alfa-beta (2)

- **Profundizamiento iterativo o progresivo** (Slate y Atkin)
- Muy utilizado en juegos en los que el tiempo tiene un papel importante de restricción.
- Consiste en estudiar hasta una determinada profundidad *p*. Si queda tiempo, se estudian *k* niveles más (profundidad *p*+*k*). Así hasta que ya no quede más tiempo.
- Aunque pueda aparecer que se pierde mucho tiempo cada vez que se baja un nivel, por tener que recalcular el árbol, se ha demostrado que no hay tal pérdida de tiempo debido a la ordenación de los nodos resultante de las iteraciones anteriores.

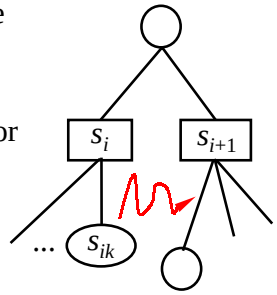
Otras mejoras del alfa-beta (3)

- **Movimiento nulo** (Goetsch y Campbell)
- Método para ayudar a la poda del árbol Alfa-Beta.
- En los nodos MAX, se evalúa la situación de ese nodo o, lo que es lo mismo,se evalúa la situación correspondiente a no mover.
- Esa evaluación representa un límite inferior de lo que va a devolver ese nodo, puesto que va a ser la peor jugada. Por tanto, si ese valor es mayor o igual que el β del nodo padre, se puede podar sin estudiar ninguno de sus hermanos.
- Al tipo de situaciones en las que mover es la peor jugada se les denomina, en el argot, posiciones "zugzwang".



Otras mejoras del alfa-beta (4)

- **Movimiento asesino (**Huberman)
- También busca incrementar el número de podas.
- Si en un determinado nodo s_i, se han estudiado sus sucesores y el mejor sucesor ha sido s_{ik}, en el siguiente nodo hermano de s_i, s_{i+1}, se intenta estudiar primero la jugada correspondiente a la s_{ik}.
- Si fue la mejor en el nodo hermano, la heurística dice que es muy posible que también sea la mejor en el nodo s_{i+1}.



Otras mejoras del alfa-beta (5)

- **Ventana**
- En lugar de empezar con la ventana inicial ($\alpha=m$, $\beta=M$), puede comenzarse con una ventana más pequeña.
- Esto hace que se produzca mayor número de podas, con lo que el sistema puede profundizar más.
- Si la ventana es demasiado pequeña se pueden estar desechando ramas importantes para el resultado final.

Otras mejoras del alfa-beta (6)

- **Movimientos de libro**: se trata de tener un catalogo de jugadas para cada configuración del tablero. Para el juego completo es imposible, pero se puede tener un catalogo para aperturas y otro para finales. Reservando la exploración alfabeta para el desarrollo intermedio del juego.

Azar

- Todavía es posible aplicar la exploración minimax pero teniendo en cuenta la presencia de nodos aleatorios.