Tema 6. Ramifica y Poda

Tabla de Contenidos

Introducción

- Aplicación para resolución de problemas de:
  - Optimización
  - Juegos
- Técnica similar a Backtracking que se basa en el análisis del árbol de estados de un problema
  - Puede interpretarse como una generalización o mejora del Backtracking
  - Realiza un recorrido sistemático del árbol
  - El recorrido no necesariamente se hace en profundidad
- Complejidad de orden exponencial
  - Su aplicación a problemas muy grandes ha demostrado ser eficiente mejorando a la técnica de backtracking

Ramifica y Poda

Introducción

- Se plantea una estrategia de **ramificación**
- Se plantean técnicas de **poda** para eliminar los nodos que no lleven a soluciones óptimas
- Se plantean **métodos de estimación del beneficio** máximo que puede obtenerse partiendo de un nodo
- En base a la estimación se ejecuta la **poda**

Ramifica y Poda vs. Backtracking

- En Backtracking en cuanto se genera un nuevo nodo hijo pasa a procesarse (estudiarse)
  - Búsqueda y exploración en profundidad
- En Ramifica y Poda, se generan todos los nodos hijos del nodo actual y después se van procesando
  - Búsqueda y exploración no necesariamente en profundidad
- En Backtracking los únicos nodos vivos son los que están en el camino desde la raíz hasta el nodo que se está estudiando
- En Ramifica y Poda puede habar más nodos vivos que se almacenan en la ***lista de nodos vivos***

Ramifica y Poda vs. Backtracking

- Técnicas de delimitación de nodos fracaso
  - Backtracking ⇒ Test fracaso que delimita el conjunto de nodos a estudiar
  - Ramifica y Poda ⇒ Cota que nos indica los nodos que pueden ser más prometedores

Método general

Estrategia

- Exploración del árbol desde la raíz (problema original)
- Estimación de cotas inferiores y superiores al problema raíz
- Si las cotas cumplen las condiciones establecidas hemos encontrado una solución optimal y el procedimiento finaliza
- Si no es así, el problema se divide en dos o más
- Se realiza la búsqueda en cada subproblema mediante aplicación recursiva del método

Método general

En un nodo *i* siempre tendremos

- Tendremos
  - Una cota superior **CS(i)** del beneficio (o coste) que podemos obtener a partir de ese nodo
  - Una cota inferior **CI(i)** del beneficio (o coste) que podemos obtener a partir de ese nodo
  - **Estimación del beneficio** (o coste) que se puede obtener a partir de este nodo
    - Podría ser la media de los anteriores
- Podremos aplicar una poda
  - Tras recorrer varios nodos (j entre 1..n) y estimar CS y CI para cada nodo j
  - Dos casos a) y b)

Estrategia de poda en un nodo i

- **Caso a**: podemos obtener una solución válida a partir del nodo i
  - Podar el nodo si
    - $CS(i) \leq CI(j)$ para algún nodo j de los generados
    - Ejemplo: mochila con árbol binario
      - A partir de **a** $\Rightarrow$ CS(a) = 4 *estimación*
      - A partir de **b** $\Rightarrow$ CI(b) = 5 *garantía*
      - Se poda **a** sin perder ninguna solución óptima

Método general

Estrategia de poda en un nodo i

- **<u>Caso b</u>**: puede que no lleguemos a una solución válida a partir del nodo i
  - Podar el nodo si
    - CS(i) ≤ Beneficio(j) para algún nodo j que ya es solución final (factible)
    - Ejemplo: *n* reinas
      - A partir de una solución parcial no está garantizado que lleguemos a una solución

Método general

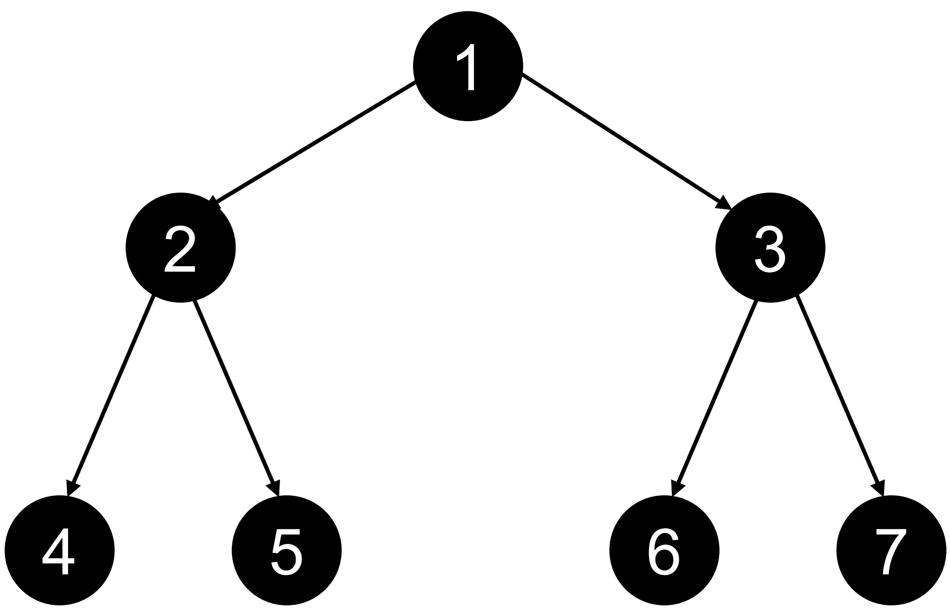
Estrategias de ramificación

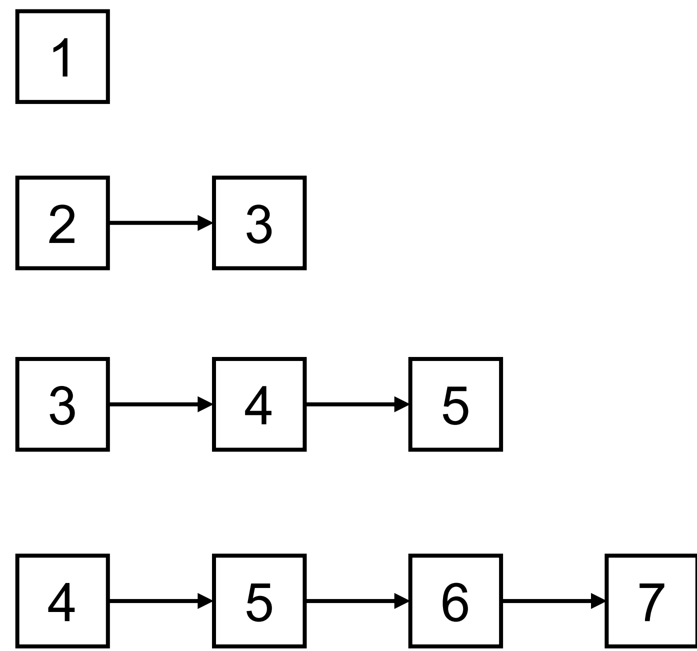
- El árbol de soluciones es implícito (no se almacena)
- Utilización de **lista de nodos vivos**
  - Nodos que se han generado pero que no se han explorado todavía
  - Nodos pendientes de tratar
  - La forma de la lista condicionará el recorrido que se va a realizar

Lista de Nodos Vivos

Cola FIFO

- Recorrido en anchura

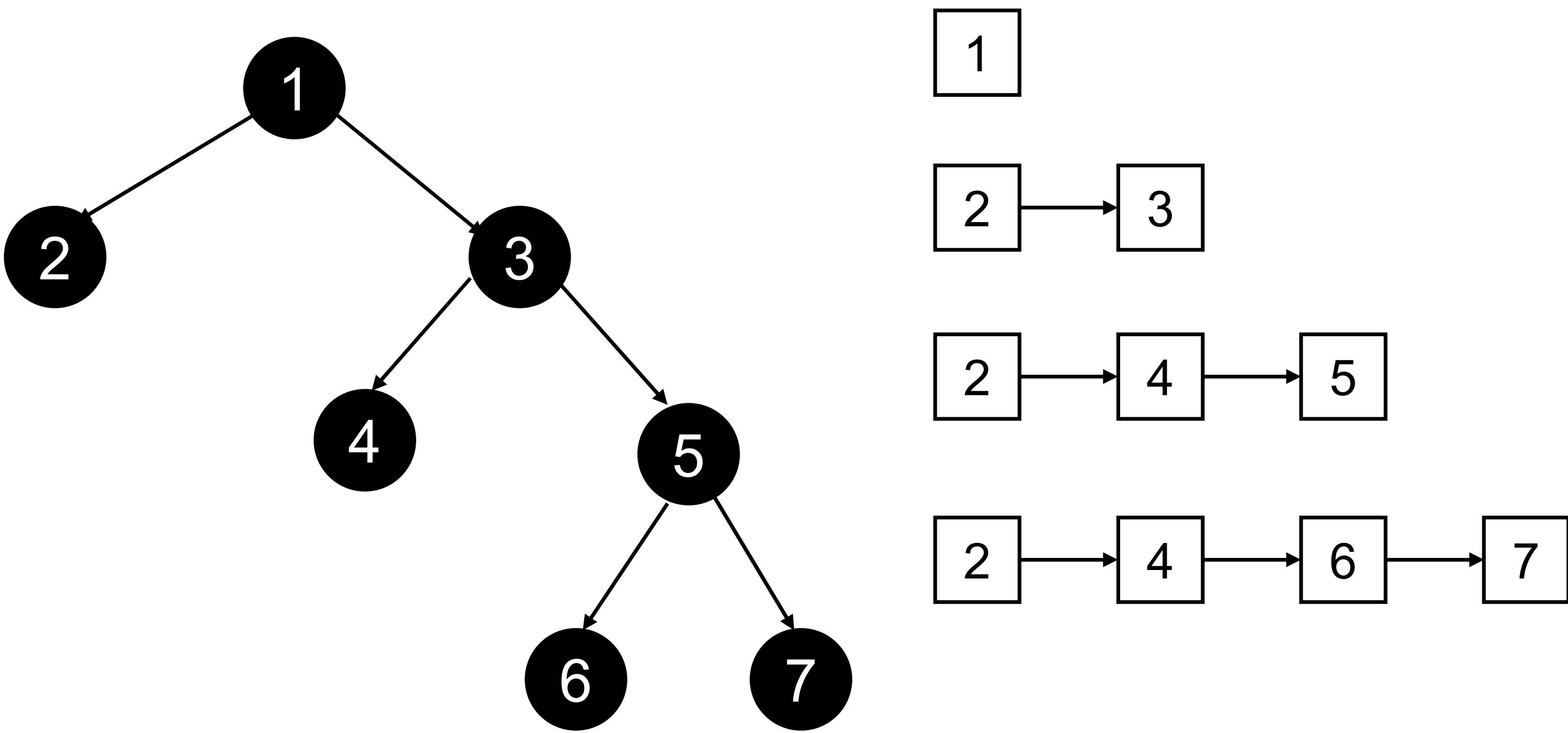




Lista de Nodos Vivos

Pila LIFO

- Recorrido en profundidad



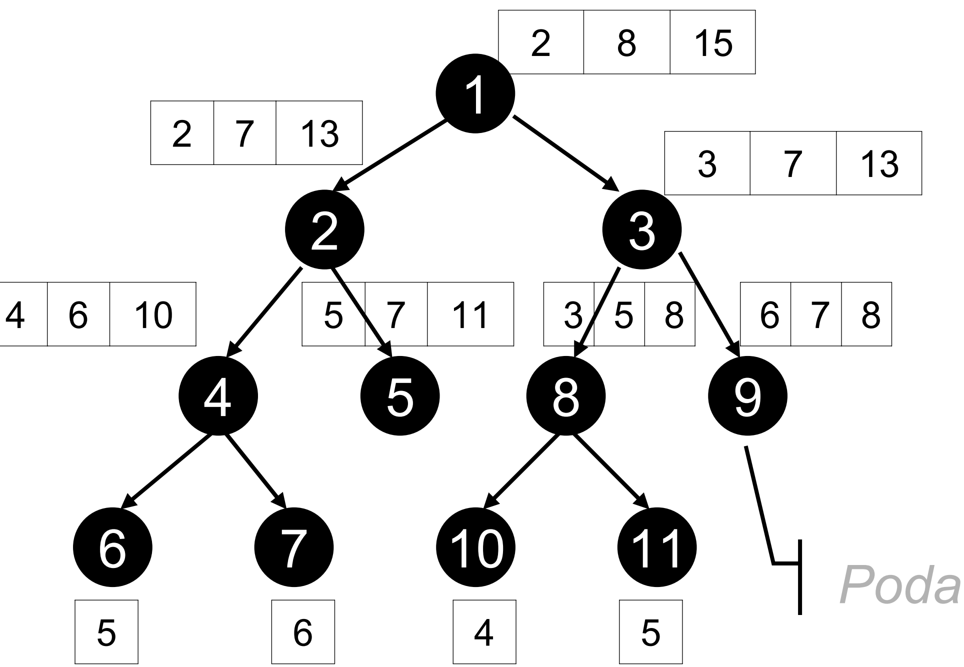
Lista de Nodos Vivos

Estrategia LC (*Least Cost*)

- Utiliza la estimación del beneficio
- Da prioridad a los nodos con mejor valor estimado
- En caso de igualdad, se puede adoptar política LIFO o FIFO convirtiéndose en:
  - LC-FIFO
  - LC-LIFO

Ejemplo LC-FIFO

- Problema de minimización
- A partir de un nodo siempre hay solución
- ***C*** ⇒ valor de las menor de las cotas superiores hasta el momento
- Si para algún nodo i, ***CI(i)≥C***, entonces podar ***i***



C	LNV
15	1
13	2 → 3
10	4 → 3 → 5
5	3 → 5
5	8 → 5
4	5

Esquema general

Consideraciones

- Problema de minimización
- Existe solución a partir de cualquier nodo

```
RamificaPoda(nodo_t raiz, solucion_t *s) {
  LNV = {raiz};          C = CS(raiz);          s = ∅;
  while ( !esVAcio(LNV) ) {
    x = seleccionar(LNV); /* criterio FIFO, LIFO, ... */   eliminar(LNV,x);
    if ( CI(x) < C )      /* si no se cumple no se trata (poda) */
      for (i=1; i<numeroHijos(x); i++) {
            y = hijo(x,i);
            if ( esSolFinal(y) && beneficio(y)>beneficio(s) ) {
              s = y;                    c = min(C,coste(y));
            } else {
              if ( !esSolFinal(y) && CI(y)<C ) {
                LNV = anyadir(LNV,y);         c = min(C,CS(y));
              }
            }
      }
  }
}
```

Método general

Eficiencia

- Parámetros importantes
  - Número de nodos recorridos y/o efectividad de la estrategia de poda
  - Procedimiento para estimación del coste/beneficio
  - Gestión de la Lista de Nodos Vivos (LNV)
- Peor caso
  - *Backtracking* + Gestión de LNV
- Caso medio
  - Mejora respecto *Backtracking*

Mejora de la eficiencia

- Árboles de gran tamaño
  - Realización de estimaciones muy precisas
  - Tiempo importante en los procesos de cálculo de estimaciones
  - Poda exhaustiva del árbol
- Árboles pequeños
  - Realización de estimaciones poco precisas
  - Poco tiempo para cada nodo
  - Poda poco significativa
- Recomendación
  - Equilibrio entre precisión en el cálculo de las cotas y su tiempo de cálculo

Consideraciones de diseño

- Representación de la solución
- Generación de descendientes

Backtracking

- Cálculo de las cotas
- Estimación del beneficio
- Estrategia de ramificación
- Estrategia de poda

Ramific
y poda

Problema de la Mochila 0/1

Representación de la solución

- Tuplas (X1, …, Xn) con Xi $\in$ {0,1}
- Árbol binario
- Hijos de un nodo
  - (X1, …, Xk, 0)
  - (X1, …, Xk, 1)

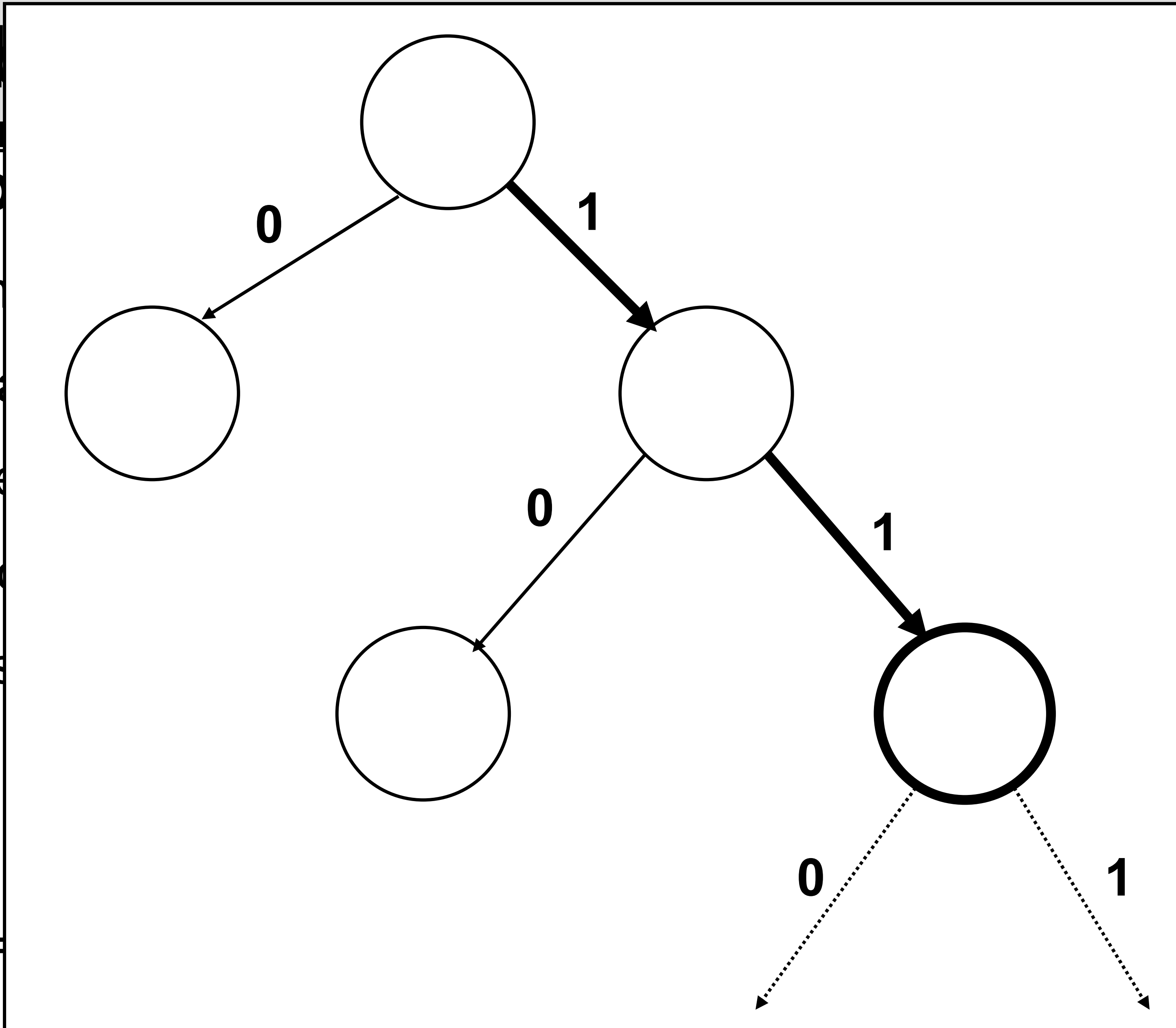
Cálculo de las cotas

- Cota inferior (CI)
  - Beneficio que se obtendría si no se incluyese ningún objeto más
- Cota superior (CS)
  - Resolución del problema a partir del nodo actual mediante un algoritmo voraz
  - Calcular el beneficio considerando solo los objetos introducidos sin fraccionar
- Beneficio (BE)
  - Suponemos los objetos ordenados de forma creciente según su densidad
  - Añadir a la solución actual el beneficio obtenido al introducir todos los objetos restantes que quepan

Prol

Ejemplo

- Núm
- Capa
- Bene
- Node
- Hijos
- CI =
- CS
- BE =



Poda

- **C** almacenará el valor de la mayor cota inferior hasta el momento
- Dado un nodo ***i***, si ***CS(i)*** $\leq$ ***C*** $\Rightarrow$ *Podar* ***i***

Ramificación

- Estrategia LC
  - Priorizar las ramas con mayor beneficio esperado
- LC-LIFO $\Rightarrow$ BE-LIFO
  - En caso de igualdad se hará un recorrido en profundidad

Problema de la Mochila 0/1


```
MochilaBB(int b[], int p[], int M, nodo_t *s) {
  inicio = NodoInicial(b,p,M);
  C = CI(inicio);          LNV = {inicio};          s.b_actual = INFINITO;
  while (! esVacio(LNV) ) {
    x = Selecion_MB_LIFO(LNV);                      LNV = LNV – {x};
    if (CS(x) > C)  /* Poda si no se cumple */
         for (i=0; i<=1; i++) {
           y = generar(x,i,b,p,M);
           if ( (y.nivel == n) && (y.b_actual > s.b_actual) ) {
                  s = y;     C = max(C, s.b_actual);
           } else {
                  LNV = LNV + {y};          C = max(C, CI(y));
           }
         }
  }
}
```

Problema de la Mochila 0/1

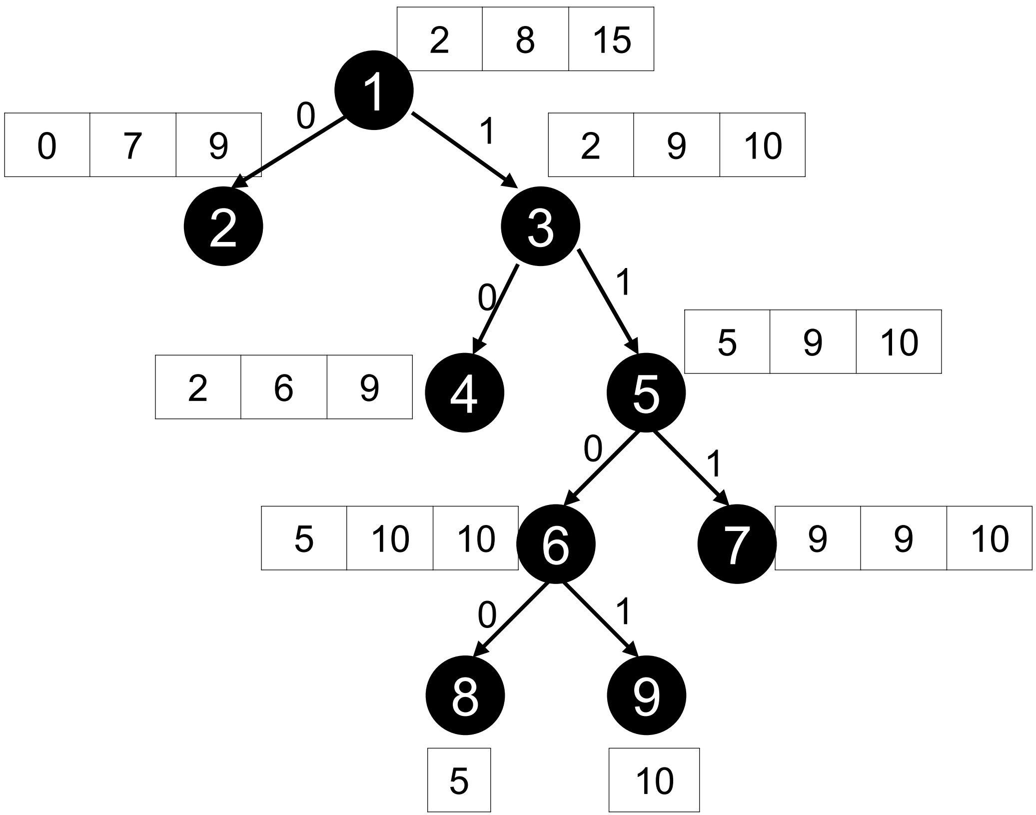

```
nodo_t *NodoInicial(int b[], int p[], int M) {
      nodo_t mi_nodo;
      mi_nodo.CI = 0;
      mi_nodo.CS = MochilaVoraz(1,M,b,p);
      mi_nodo.BE = MochilaVoraz01(1,M,b,p);
      mi_nodo.nivel = 0;
      mi_nodo.b_actual = 0;
      mi_nodo.p_actual = 0;
      return mi_nodo;
}
```

Problema de la Mochila 0/1


```
nodo_t *Generar(nodo_t x, int i, int b[], int p[], int M) {
	nodo_t mi_nodo;
	mi_nodo.tupla = x.tupla;			mi_nodo.nivel = x.nivel + 1;
	mi_nodo.tupla[mi_nodo.nivel] = i;
	if (i == 0) {
		mi_nodo.b_actual = x.b_actual;		mi_nodo.p_actual = x.p_actual;
	} else {
		mi_nodo.b_actual = x.b_actual + b[mi_nodo.nivel];
		mi_nodo.p_actual = x.p_actual + p[mi_nodo.nivel];
	}
	mi_nodo.CI = mi_nodo.b_actual;
	mi_nodo.BE = mi_nodo.CI
		+ MochilaVoraz01(mi_nodo.nivel+1,M-mi_nodo.p_actual, b,p);
	if ( mi_nodo.p_actual > M ) { /* Descartamos el nodo */
		mi_nodo.CI = mi_nodo.CS = mi_nodo.BE = INFINITO;
	}
	return mi_nodo;
}
```

Problema de la Mochila 0/1

Ejemplo: n=4, M=7, b=(2,3,4,5), p=(1,2,3,4)



C	LNV
0	1
2	3 → 2
5	5 → 2 → 4
9	6 → 7 → 2 → 4
10	7 → 2 → 4
10	2 → 4
10	4

Viajante de comercio

Planteamiento

- Conocidas las distancias entre ciudades
- A partir de una ciudad, visitar cada ciudad exactamente una vez y regresar al punto de partida
- Recorrer la menor distancia posibles

Formalmente

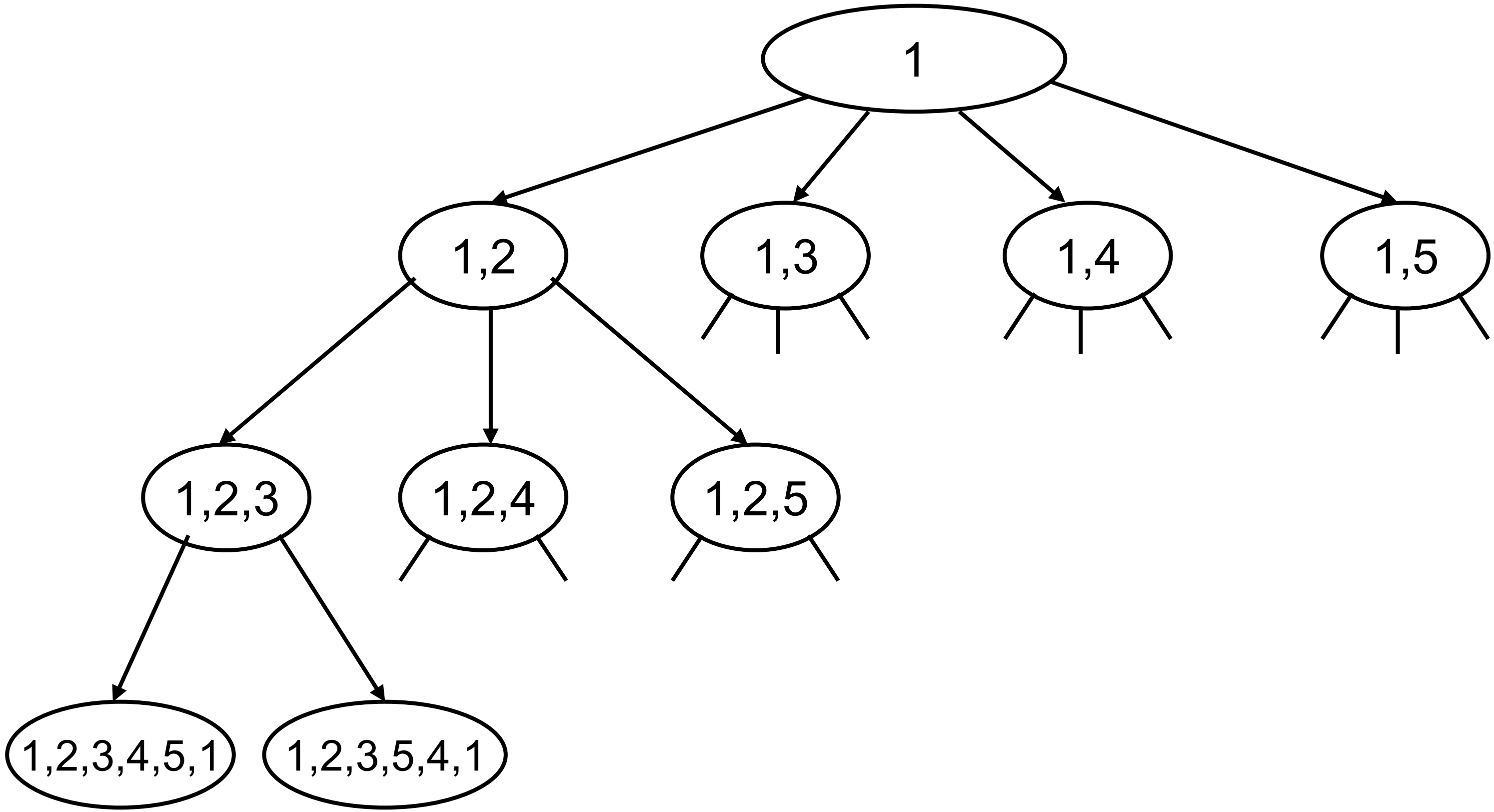
- Grafo orientado ⇒ G=(V,A)
- Longitud de (i,j)∈A ⇒ D[i,j]
- Origen del camino en la ciudad 1
- Función objetivo

$$F(x) = D[1, x_1] + D[x_1, x_2] + \ldots + D[x_{n-2}, x_{n-1}] + D[x_n, 1]$$

Árbol de expansión del problema

- Solución como secuencia de decisiones
- Un vértice en cada etapa
- Vector que indica el orden en el que se visitarán los vértices
- En cada nodo se podrán generar *N-k* hijos, ya que el resto ya han sido visitados

Árbol de expansión del problema



Otros criterios para poda

- Estimación del costo posible a acumular
- Matriz de distancias reducida (filas y columnas reducidas)
  - La cantidad *L* restada de filas y columnas es una cota inferior de longitud en un hamiltoniano de longitud mínima, por lo que sirve como estimación para el nodo raíz

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \xrightarrow{L=25} \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Planteamiento

- ***N*** trabajos han de ser realizados por ***N*** personas, y se sabe que la persona ***i*** tardará un tiempo C_{ij}, en hacer el trabajo ***j***
- Minimización del coste de la asignación total

	1	2	3	4
a	94	1	54	68
b	74	10	88	82
c	62	88	8	76
d	11	74	81	21

	1	2	3	4	5
a	11	17	8	16	20
b	9	7	12	6	15
c	13	16	15	12	16
d	21	24	17	28	26
e	14	10	12	11	15

Asignación de tareas

Representación de la solución

- Vector cuyo *k-ésimo* elemento indica la tarea asignada a la persona *k*

Árbol de expansión del problema

- Secuencia de decisiones (una en cada etapa o nivel) según la estructura que va a tener la solución al problema
  - En cada paso decidiremos qué tarea asignamos a una persona, eligiendo entre las que no estén asignadas
- Para cada nodo se generarán *N-k* hijos como máximo
  - Poda elemental

Asignación de tareas

Poda mediante función de coste (LC)

- Problema de minimización
- Necesitamos una cota inferior (teórica y no necesariamente alcanzable)
- Cálculo de los mínimos de los elementos aún no asignados de cada columna de C
  - Independientemente de a quién se la asignemos, constituyen los mejores valores que vamos a poder alcanzar para cada trabajo
- Dado un nodo ***i***, si ***CI(i)*** $\geq$ **C** $\Rightarrow$ *Podar* ***i***

Introducción

- Existen muchos juegos de estrategia en los que se enfrentan dos jugadores y uno de ellos puede conseguir el objetivo (ganar al otro)
- La estrategia del juego se puede representar mediante un grafo dirigido
  - **Nodo**: situación particular del juego
  - **Arista**: jugada o movimiento válido entre dos situaciones (nodos)
- El grafo podría llegar a ser infinito

Aproximaciones y consideraciones

- El juego se desarrolla entre dos jugadores que se van turnando
- Juego **simétrico**
  - Las reglas son las mismas para ambos jugadores
- Juego **determinista**
  - la casualidad no interviene en el resultado
- Ningún caso del juego puede tener una duración infinita
- Ninguna situación de juego ofrece un número infinito de jugadas válidas para el jugador que tenga el turno

Representación de la estrategia del juego

- **Situación *terminal***
  - Situación en la que no existe ninguna jugada válida
  - Nodos que no poseen sucesores (*nodos hoja*)

- Asociación de etiquetas a cada nodo
  - Situaciones ***terminales***
  - Situación de ***victoria***
    - *Al menos uno de sus sucesores representa una situación de derrota*
  - Situación de ***derrota***
    - *Todos sus sucesores representan situaciones de victoria*
  - Situación de ***empate***
    - *Cualquier otra situación no terminal*
- Siguiendo el grafo se obtiene la estrategia ganadora

Otras casuísticas

- Para el desarrollo de algunos juegos es necesario almacenar información adicional de cada situación
- En algunos tipos de juegos el grafo contiene tantos nodos que es inviable explorarlo en su totalidad
  - Ejemplo: ajedrez
- Se puede plantear la exploración del grafo en las proximidades de la situación actual para ver cómo puede evolucionar la situación actual
  - *Heurística **MiniMax***
- Estrategias de búsqueda y exploración
  - En *anchura* o en *profundidad*

Algoritmo MiniMax

Planteamiento y objetivos

- Descartar el estudio exhaustivo del grafo que representa la estrategia
- Realización de una búsqueda parcial en torno a la situación actual
- Búsqueda de una jugada que se prevé entre las mejores
- Maximizar la optimalidad de sus posiciones y minimizar las del contrario
- Se utilizan heurísticas para evaluar las posiciones y alternativas

Estrategia general

- Definición de una función estática de evaluación de atribuya un valor a cada posible situación accesible desde la posición actual
  - Dependiendo del juego la función puede tener en cuenta diversos factores
  - Compromiso entre precisión y tiempo de cálculo
  - Como heurística, proporciona una jugada que se encuentra entre las mejores disponibles
- Valores de la función
  - Valor positivo alto $\Rightarrow$ Gana el jugador A
  - Valor negativo alto $\Rightarrow$ Gana el jugador B
  - Valor próximo cero o cero $\Rightarrow$ Empate

MAXIMIZAR

Algoritmo MiniMax

Estrategia general

- Generar árbol desde la posición actual hasta los nodos terminales
- Aplicar la función de evaluación a los nodos terminales
- Propagar hacia arriba el resultado para obtener valores para todos los nodos
  - Maximizando para jugador **MAX**
  - Minimizando para jugador **MIN**
- Elección de la jugada que proporcione un valor máximo de la función de evaluación

Algoritmo MiniMax

Estrategia general

- *MINIMAX-VALOR*(n)
  - *FUNCION_EVALUACION*(n) Si *n* es nodo terminal
  - $\max_{s \to sucesor(n)} MINIMAX-VALOR(s)$ Si *n* es un nodo **MAX**
  - $MIN_{s \to sucesor(n)} MINIMAX-VALOR(s)$ Si *n* es un nodo **MIN**

Algoritmo MiniMax

Aplicación

- Evaluación de la situación actual de un juego (estado) considerando las posibles jugadas

- Decisión de la mejor jugada a partir del estado del juego

Algoritmo MiniMax

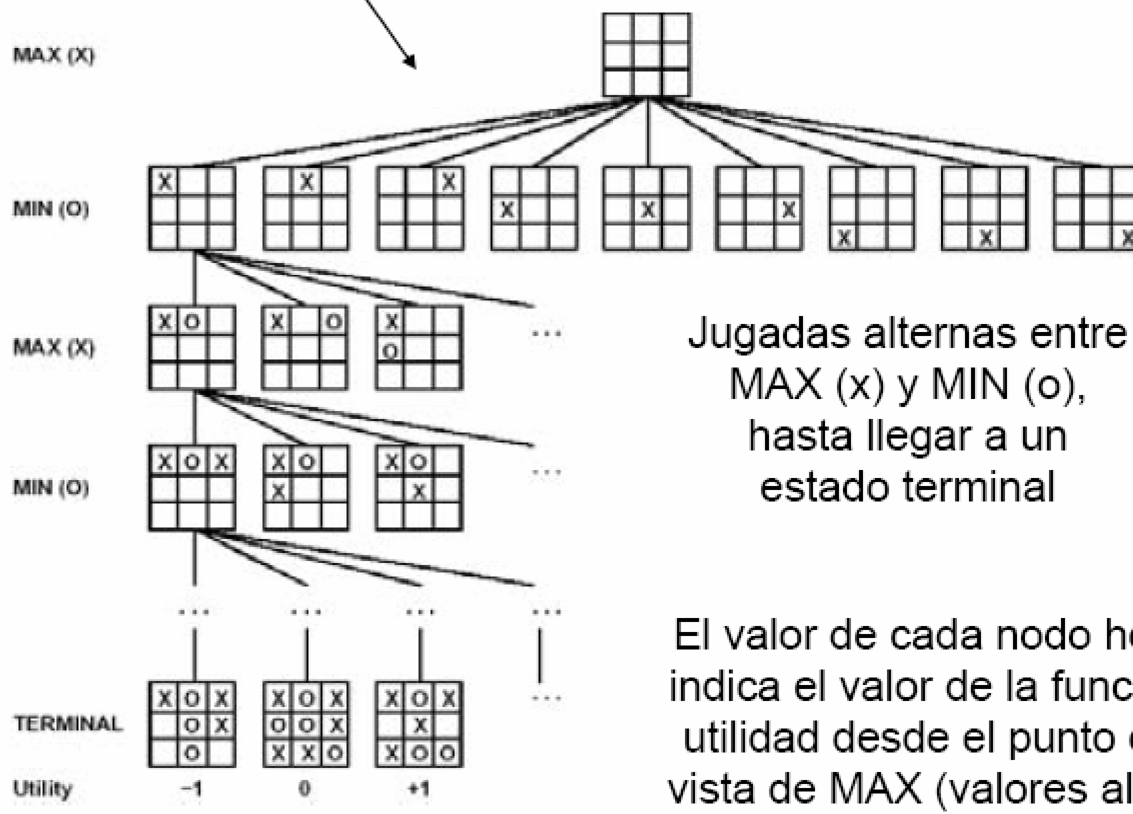
Aproximaciones

- **Trivial**
  - Generar todo el árbol de jugadas y decidir
  - Etiquetar jugadas terminales con un valor de utilidad (+1,-1,0)
  - Búsqueda de una serie de movimientos que den como ganador a un jugador (***MAX***)
  - Propagación de los valores de las jugadas terminales hasta la raíz
  - En juegos complejos es irrealizable

Algoritmo MiniMax

Ejemplo: tres en raya

Inicialmente MAX puede realizar uno de entre nueve movimientos posibles

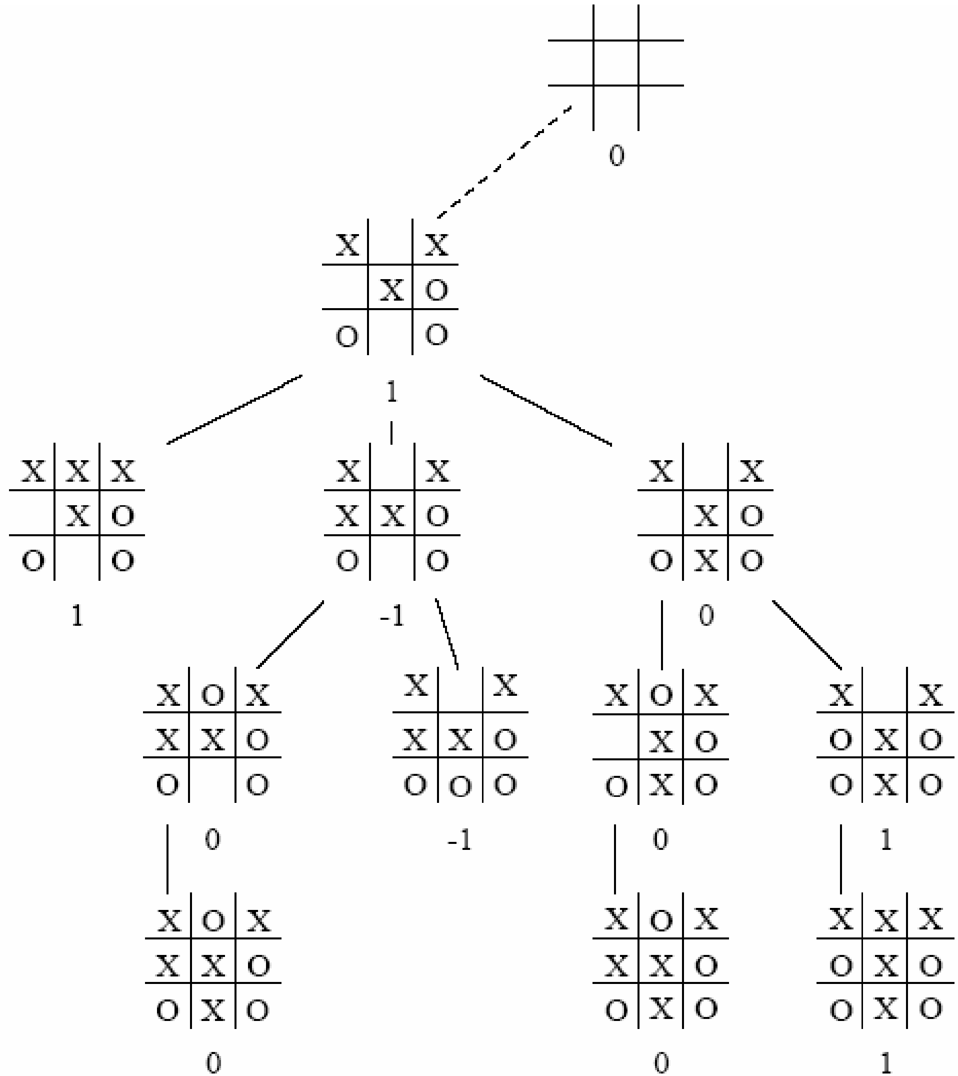


Jugadas alternas entre MAX (x) y MIN (o), hasta llegar a un estado terminal

El valor de cada nodo hoja indica el valor de la función utilidad desde el punto de vista de MAX (valores altos son buenos para MAX y bajos buenos para MIN)

Algoritmo MiniMax

Tres en Raya



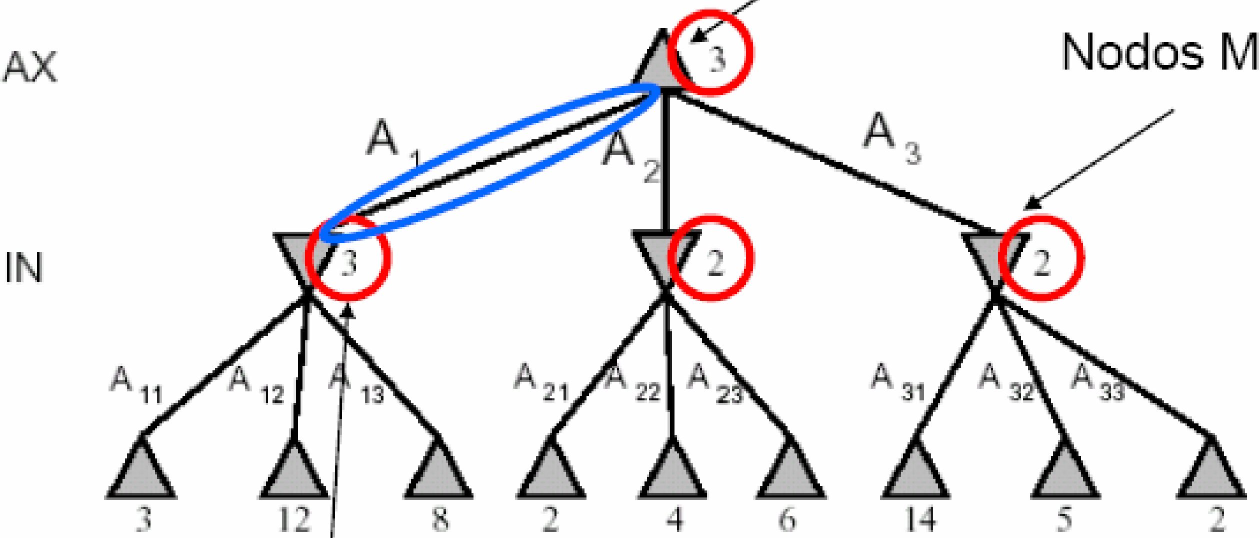
Algoritmo MiniMax

Aproximaciones

- **Heurística**
  - Definición de una función que indique lo cerca que está un estado de otros que representan **Victoria** (o **Derrota**)
    - Función ***MINIMAX-VALOR(n)***
      - ✓ $+\infty \rightarrow$ *ganar*
      - ✓ $-\infty \rightarrow$ *ganar*
    - Utilización de información del dominio
  - Algoritmo que realiza una búsqueda en profundidad *limitada*

Algoritmo MiniMax

Nodos MAX, le toca mover a MAX



La mejor jugada de **MAX** es **A1** porque genera el mayor valor MINIMAX entre sus sucesores

La mejor jugada de **MIN** es **A11** porque genera el MENOR valor MINIMAX entre sus sucesores

Valores de la función de utilidad para MAX

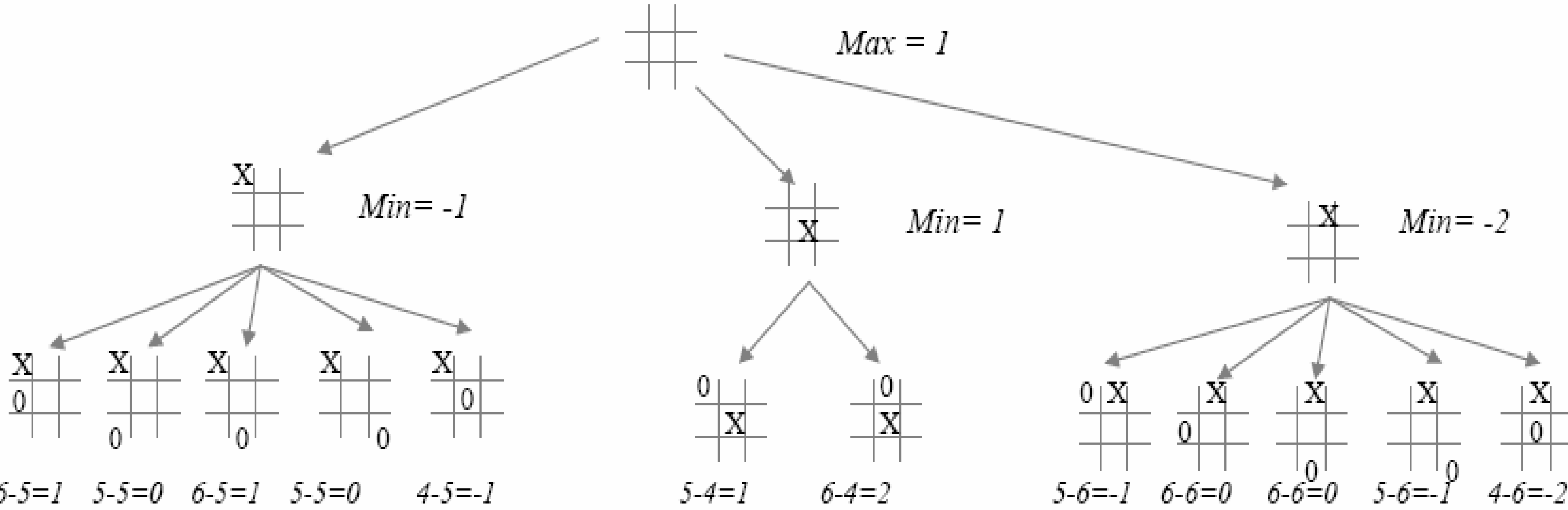
Valores minimax (cada nodo tiene asociado valor minimax o MINIMAX-VALUE(n))

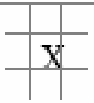
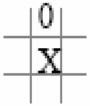
Aproximaciones

- **Heurística**: Ejemplo de las tres en raya
  - ***MAX*** *juega con* ***X***
  - ***MIN*** *juega con* ***O***
  - ***Utilidad(n)*** *=* ***e*** *= (#filasX + #columnasX + #diagonalesX) - (#filasO + #columnasO + #diagonalesO)*
    - *líneas completas disponibles*
  - En el ejemplo, utilización de *profundidad 2* como condición de parada

Algoritmo MiniMax

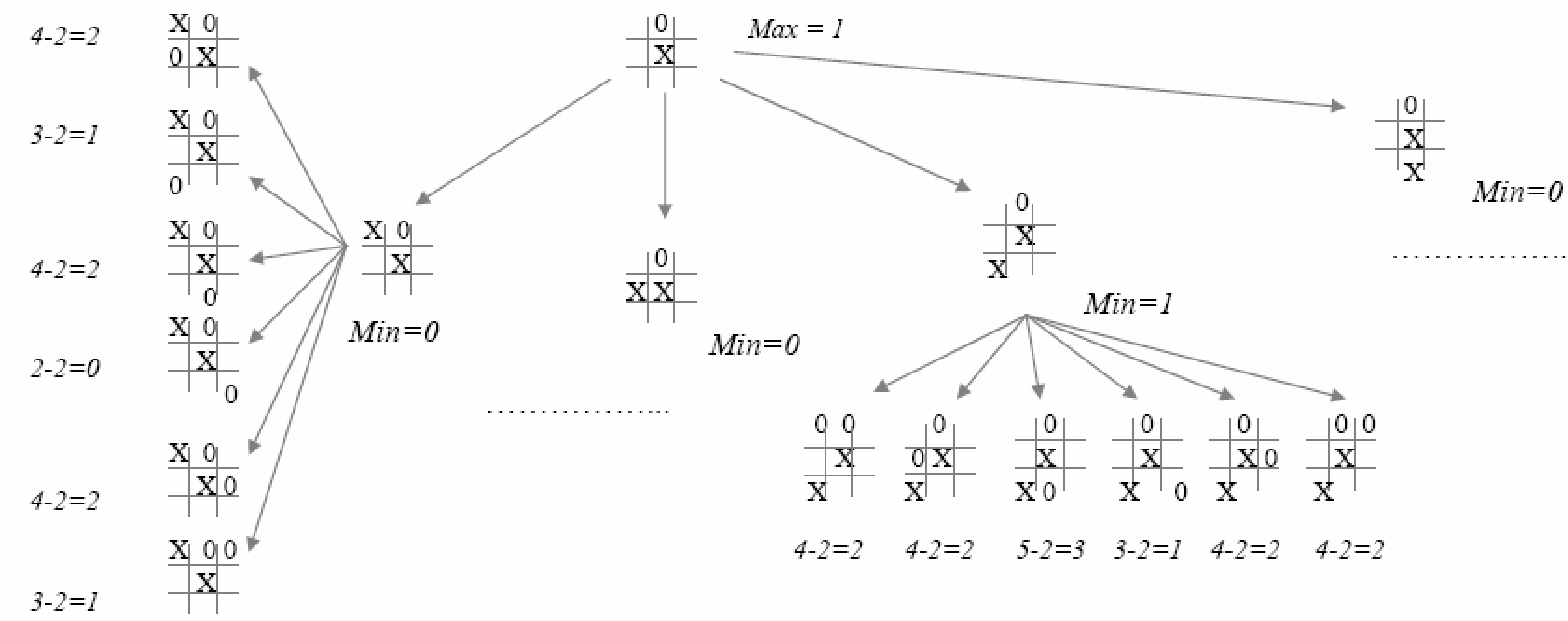
Tres en raya

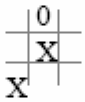
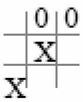


La mejor jugada de max es pues  *tras lo cual min podría jugar* 

Algoritmo MiniMax

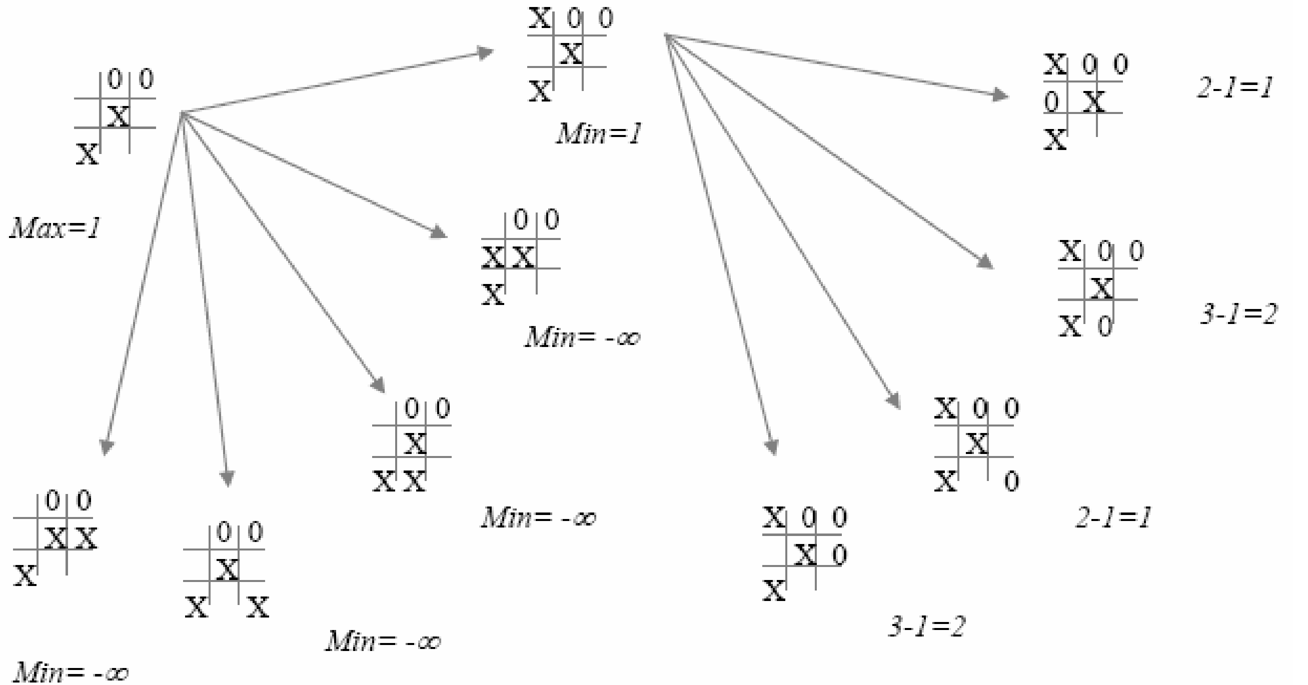
Tres en raya

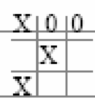


La mejor jugada de max es pues  *tras lo cual min podría jugar* 

Algoritmo MiniMax

Tres en raya



La mejor jugada de max es pues 

Algoritmo MiniMax

Algoritmo general

```
float MINIMAX(tablero_t B, modo_t modo) {
/* Evalúa la utilidad del tablero B, en el supuesto de que es el movimiento del jugador
1 si modo=Max o es movimiento del jugador 2 si modo=Min */
  tablero_t C; /* un hijo de B */      float valor; /* Valor min o max temporal */
  if (esHoja(B) return (utilidad(B));
  else {  /* asigna el valor inicial mín o máx de los hijos */
    if (equals(modo,"Max")) valor = - INFINITO;
    else valor = INFINITO;
    for (i=0; i<numHijos(B); i++) {
      C = hijo(i,B);
      if (equals(modo,"Max")) valor=max(valor,MINIMAX(C,"Min"));
      else valor = min(valor,MINIMAX(C,"Max"));      }
  }
  return (valor);
}
```

Las Tres en Raya


```
void Computador(tablero_t *T, int *mejor, int *valor) {
  int n_s, i, valor_r;
  if (esLlenoTablero(T) return *valor = EMPATE;
  else {
    if (CompGanaEnUnPaso(T,valor)) *valor = GANA;
    else {
       *valor = PIERDE;
       for (i=1; i<=9; i++)
          if(esVacioTablero(T,i)) {
            Coloca(T,i,"X");   Hombre(T,n_s,&valor_r);    DesColoca(T,i);
            if (valor_r > *valor) {
               *valor = valor_r;     *mejor = i;
            }
          }
    }
  }
}
```

Las Tres en Raya

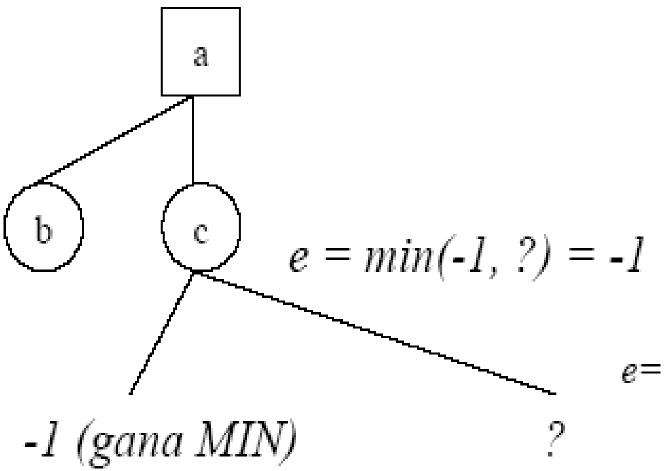

```
void Hombre(tablero_t *T, int *mejor, int *valor) {
  int n_s, i, valor_r;
  if (esLlenoTablero(T) return *valor = EMPATE;
  else {
    if (HombreGanaEnUnPaso(T,valor)) *valor = PIERDE;
    else {
       *valor = GANA;
       for (i=1; i<=9; i++)
          if(esVacioTablero(T,i)) {
            Coloca(T,i,"O");   Computador(T,n_s,&valor_r);    DesColoca(T,i);
            if (valor_r < *valor) {
               *valor = valor_r;     *mejor = i;
            }
          }
    }
  }
}
```

Planteamiento general

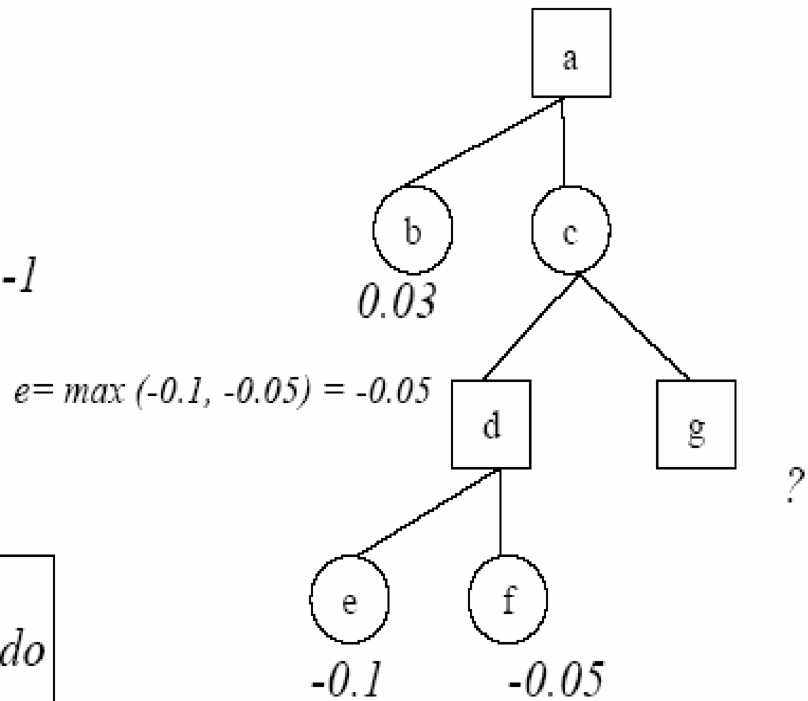
- Problema de la estrategia MiniMax
  - Examina un número elevado de nodos
  - Si la estrategia se lleva hasta el final se consiguen algoritmos poco eficientes

- Aplicación de técnicas de “Ramifica-Poda” para mejorar la situación anterior
  - En este caso es ligeramente diferente a lo visto anteriormente
  - Tratamiento de dos jugadores (MAX y MIN) con objetivos diferentes
  - Necesidad de dos cotas para los criterios de poda
  - Dependiendo del jugador al que le toque realizar el movimiento se considerará una u otra cuota

Poda Alpha-Beta



No tiene sentido seguir buscando los otros descendientes de c.



En c: *e= min(-0.05, v(g))*
por lo tanto en a: *e = max(0.03, min(-0.05, v(g))) = 0.03*
Se pueden pues podar los nodos bajo g;
no aportan nada.

Poda Alpha-Beta

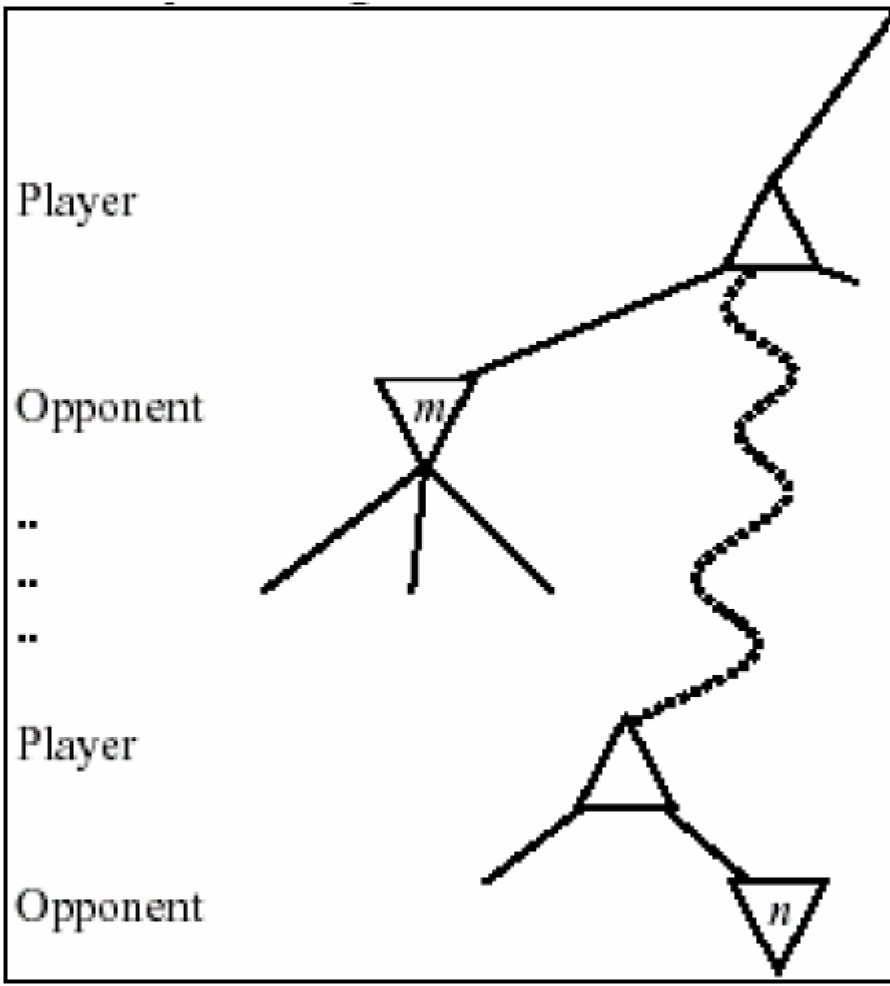
Planteamiento general

- La exploración de ciertas ramas de un árbol se puede abandonar con antelación si se dispone suficiente información sobre las mismas para determinar que no van a influir sobre las zonas más altas del árbol (*Ramificación y Poda*)
- Cotas: **alpha** y **beta**
- Dependiendo del jugador al que le toque realizar el movimiento se aplicará:
  - Poda-α
  - Poda-β

Poda Alpha-Beta

Estrategia

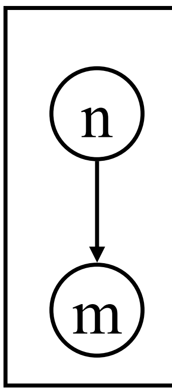
- El algoritmo efectuará un búsqueda en profundidad
- Si durante la misma se de determina que ***m*** es mejor que ***n*** para un jugador, entonces nunca se alcanzará ***n*** en el juego



Fundamentos del algoritmo

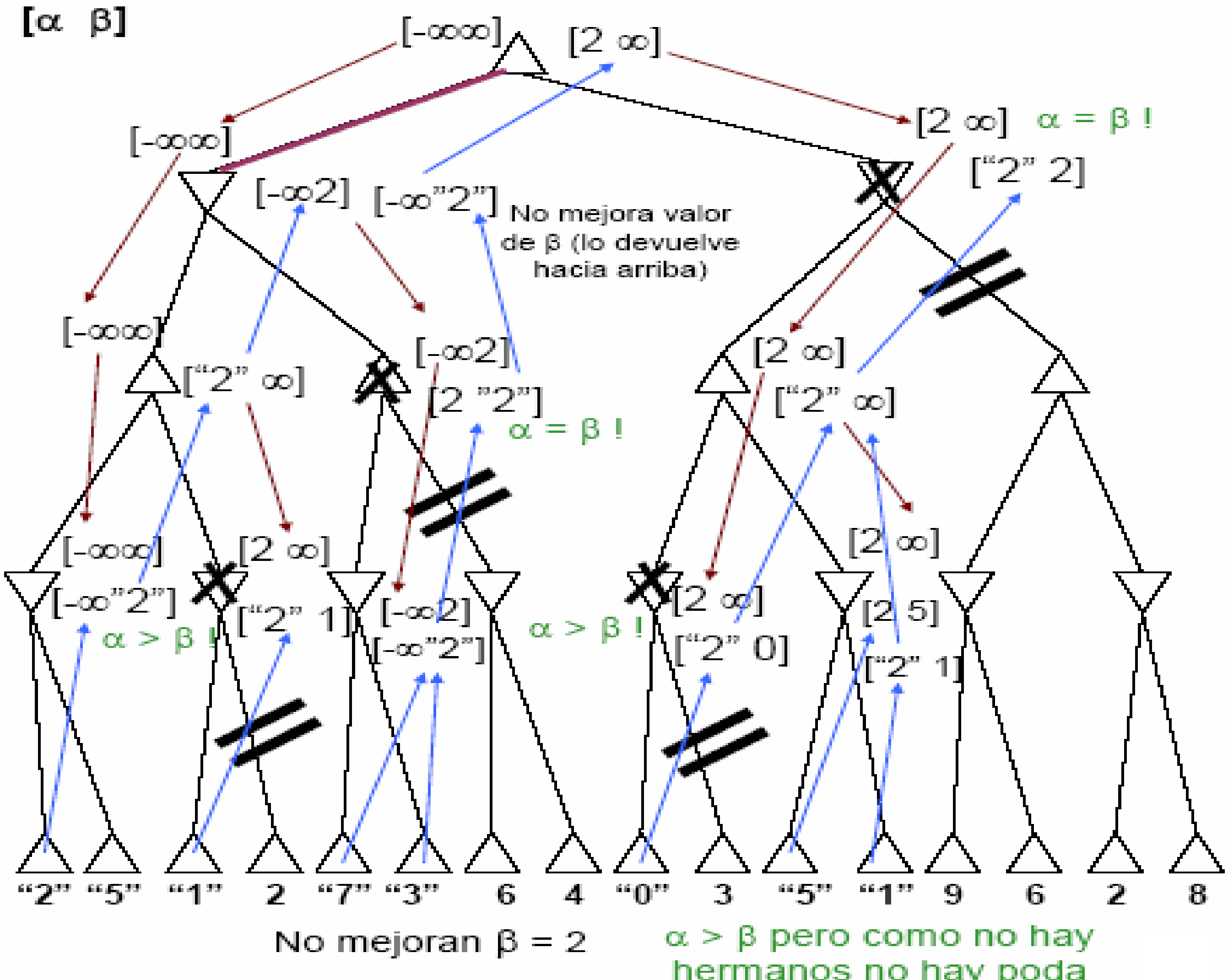
- Definimos cotas para los valores obtenidos en la propagación hacia arriba
  - Valor α como cota inferior
  - Valor β como cota superior

- Poda del nodo ***m*** con ***n*** ascendiente de ***m***
  - ***n*** nodo MAX, ***m*** nodo MIN
    - El valor de α se alcanza en nodo hijo de ***n***
    - $\alpha(n) \geq \beta(m)$
  - ***n*** nodo MIN, ***m*** nodo MAX
    - El valor de α se alcanza en nodo hijo de ***n***
    - $\alpha(m) \geq \beta(n)$



Poda Alpha-Beta

Ejemplo



Algoritmo MiniMax (α-β)


```
float MINIMAX(tablero_t B, modo_t modo, int alpha, int beta) {
  tablero_t ; /* un hijo de B */  float valor; /* Valor min o max temporal */ float temp;
  if (esHoja(B) return (utilidad(B));
  else {     /* asigna el valor inicial mín o máx de los hijos */
    if (equals(modo,“Max”)) valor = alpha;
    else valor = beta;
    for (i=0; i<numHijos(B); i++) {
      C = hijo(i,B);
      if (equals(modo,“Max”)) {
           temp = MINIMAX(C, ”Min”,valor,beta);
           if (temp<beta) valor = max(valor,temp);
           else return(temp);
      } else {
           temp = MINIMAX(C, ”Max”,alfa,valor);
           if (temp>alfa) valor = min(valor,temp);
           else return(temp);
      }
    }
  }
  return (valor);
}
```

Tres en Raya (α-β)


```
void Computador(tablero_t *T, int *mejor, int *valor, int alfa, int beta) {
  int n_s, i, valor_r;
  if (esLlenoTablero(T) return *valor = EMPATE;
  else {
    if (CompGanaEnUnPaso(T,valor)) *valor = GANA;
    else {
       *valor = alfa;          i = 1;
       while ( (i<=9) && (*valor<beta) ) {
          if(esVacioTablero(T,i)) {
            Coloca(T,i,"X");   Hombre(T,n_s,&valor_r,*valor,beta);    DesColoca(T,i);
            if (valor_r > *valor) {
               *valor = valor_r;     *mejor = i;
            }
          }
          i++;
       }
    }
  }
}
```

Tres en Raya (α-β)


```
void Hombre(tablero_t *T, int *mejor, int *valor, int alfa, int beta) {
  int n_s, i, valor_r;
  if (esLlenoTablero(T) return *valor = EMPATE;
  else {
    if (HombreGanaEnUnPaso(T,valor)) *valor = PIERDE;
    else {
       *valor = beta;          i = 1;
       while ( (i<=9) && (*valor>alfa) ) {
          if(esVacioTablero(T,i)) {
            Coloca(T,i,"O");  Computador(T,n_s,&valor_r,alfa,*valor);    DesColoca(T,i
            if (valor_r < *valor) {
               *valor = valor_r;     *mejor = i;
            }
          }
          i++;
       }
    }
  }
}
```