Búsqueda

Eduardo Morales, Enrique Sucar

INAOE

Contenido

Introducción

La solución de problemas está asociado a la inteligencia.
El proceso "normal" de solución de problemas involucra:

- Identificación y definición del problema
- Identificación del criterio de evaluación
- Generación de alternativas
- Búsqueda de una solución y evaluación
- Selección de opción y recomendación
- Implementación

En IA la solución es principalmente búsqueda y evaluación.

Búsqueda

- La búsqueda es necesaria en la solución de problemas y normalmente involucra introducir *heurísticas*.
- Las heurísticas son criterios, métodos, o principios para decidir cuál de varias alternativas de acción promete ser la más efectiva para cumplir con una meta.
- Representan un compromiso entre: (i) simplicidad y (ii) poder discriminatorio entre opciones buenas y malas.
- Las heurísticas no garantizan la acción más efectiva, pero muchas veces lo hacen.
- En problemas complejos, las heurísticas juegan un papel fundamental para reducir el número de evaluaciones y para obtener soluciones dentro de restricciones de tiempo razonables.

El problema de las 8 reinas

- Una posibilidad es lograr una solución *incremental*, acercándose a la meta poco a poco siguiendo una serie de *decisiones locales*
- Hay que asegurarse que la secuencia de transformaciones sea *sistemática*, para (i) no generar configuraciones repetidas y (ii) no excluir configuraciones deseables.
- Una forma de sistematizar la búsqueda es *construyendo* más que transformando configuraciones
- *El hecho de que podamos sistematizar la búsqueda de esta forma es que no nos podemos recuperar de violaciones a restricciones mediante operaciones futuras*.

El problema de las 8 reinas

Posibles candidatos de heurísticas:

1. Preferir colocar reinas que dejen el mayor número de celdas sin atacar. En el ejemplo: heu1(A)=8, heu1(B)=9, heu1(C)=10.
2. Ver cuál es el menor número de celdas no atacadas en cada renglón y escoger la que su número menor sea mayor. En el ejemplo: heu2(A)=1, heu2(B)=1, heu2(C)=2

El problema de las 8 reinas

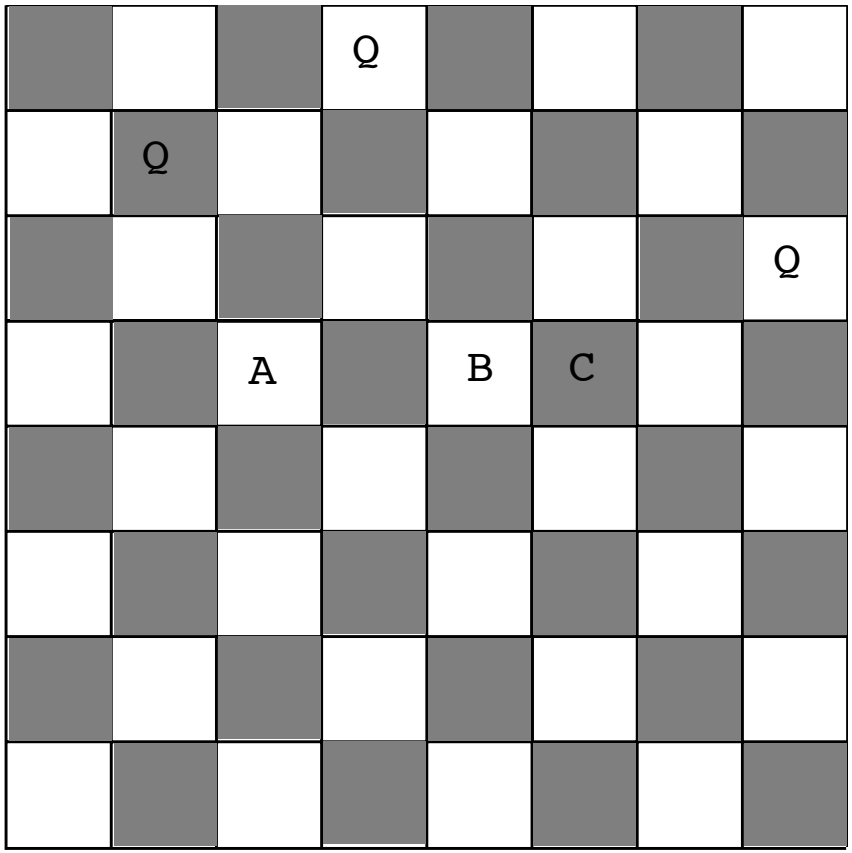


Figure: El problema de las 8 reinas

El problema del 8-puzzle

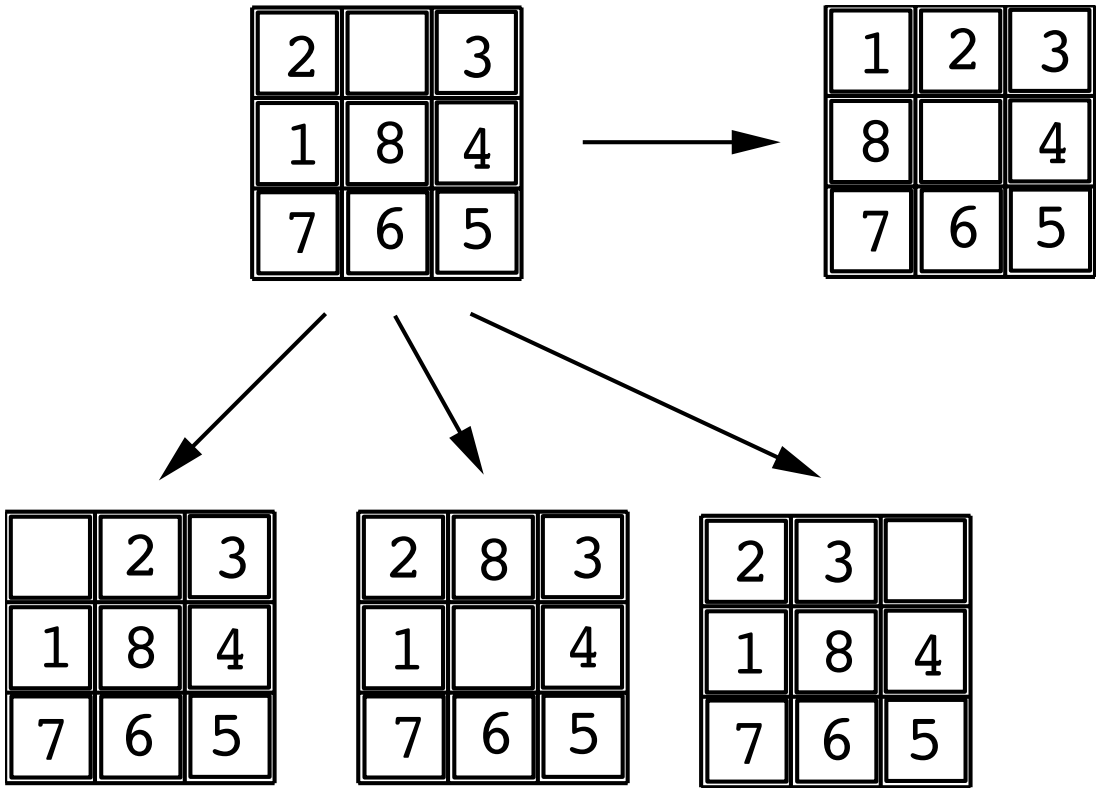


Figure: El problema del 8 puzzle

El problema del 8-puzzle

- Hacer *búsqueda exhaustiva* es impráctico
- Una posible regla es estimar *qué tan cerca se está de la solución*.
- Algunas heurísticas que se han usado para ésto son:
  1. Contar el número de cuadros que no corresponden a la meta (sin contar el blanco). En el ejemplo: heu1(A)=2, heu1(B)=3, heu1(C)=4.
  2. La suma de la distancia Manhattan de los cuadros que no corresponden a su lugar. En el ejemplo: heu2(A)=2, heu2(B)=4, heu2(C)=4.
  3. Distancia del cuadro blanco al primer cuadro fuera de su lugar. En el ejemplo: heu3(A)=1, heu3(B)=1, heu3(C)=3.
  4. Distancia entre la posición final del blanco y su posición actual. En el ejemplo: heu4(A)=2, heu4(B)=0, heu4(C)=2.

Encontrar la ruta más corta entre dos ciudades

- Dado un mapa con varias ciudades encontrar el camino más corto entre un par de ciudades.
- Pero si en lugar del mapa se nos dá informacíon de las distancias entre ciudades no es obvia la preferencia.
- Esto es porque en el mapa es posible estimar las distancias Euclideanas.
- Sin embargo, se pueden calcular las distancias a partir de las coordenadas de las ciudades, por lo que se puede usar esa *información extra* para determinar que acción tomar, en base a una estimación de lo que falta mas la distancia recorida.

El problema del agente viajero

Encontrar el circuito más corto entre ciudades.

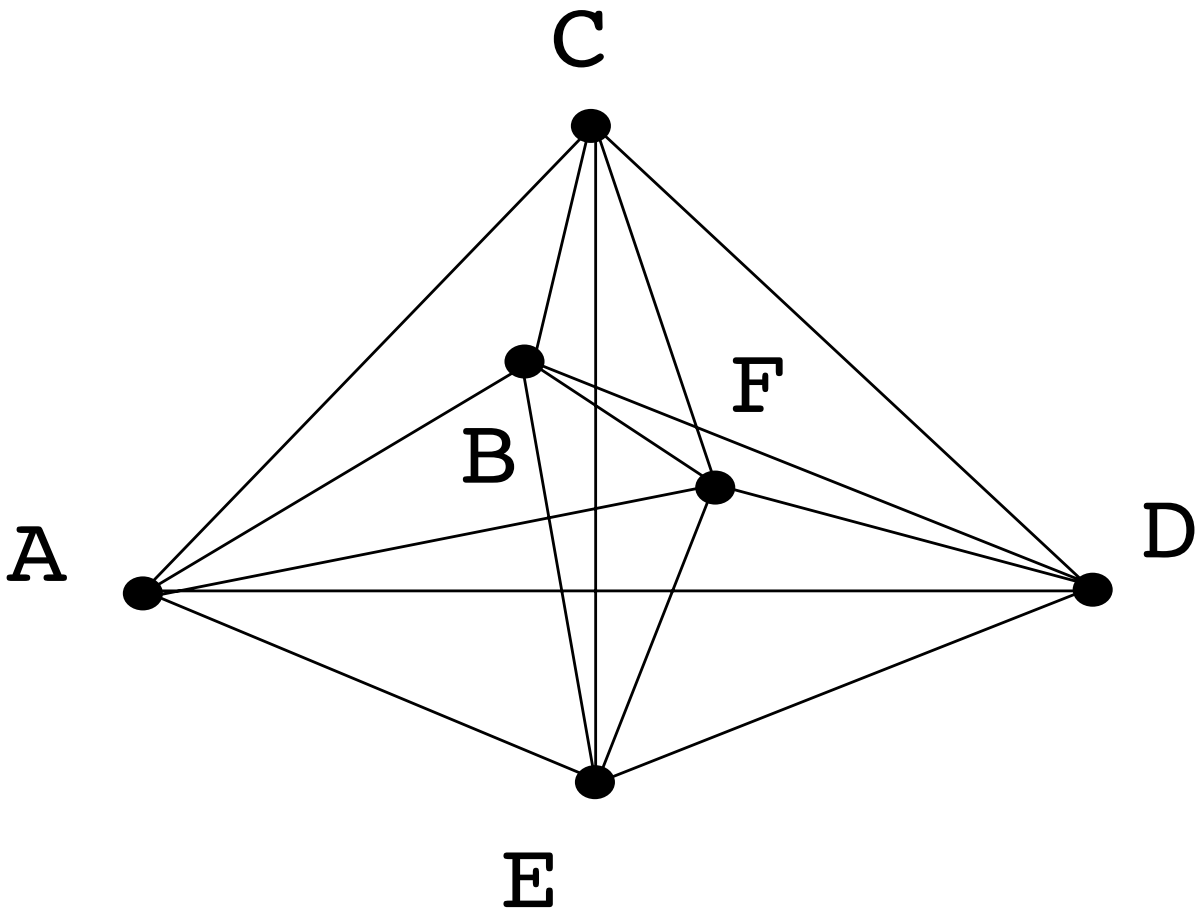


Figure: El problema del agente viajero

El problema del agente viajero

- Es un problema NP-duro, por lo que se requiere de *heurísticas adecuadas que acoten* el problema.
- Si tenemos un pedazo de ruta, de las diferentes alternativas que existen en esa ruta la mejor es la que nos estime completar la menor ruta.
- Esa estimación puede ser difícil de hacer, pero si es *optimísta* (subestima consistentemente el costo real) entonces el que nos de la mejor estimación es el mejor.

Detectar la moneda falsa (12 monedas, pesando 3 veces)

- Supongamos que tenemos una moneda falsa (más/menos pesada) en un grupo de 12, una balanza y tres intentos para detectarla.
- Se requiere de una *estrategia*, o sea una descripción de qué hacer primero y en base al resultado obtenido qué hacer después y así sucesivamente.
- Cada acción requiere hacer una estimación de qué hacer para las 3 posibles situaciones. Por lo que el mérito de una acción depende de la combinación de los méritos de las 3 posibles acciones (*regla de combinación*).
- Una vez que se decide alguna alternativa, se necesita especificar qué subproblema tiene que ser considerado después.

Representación

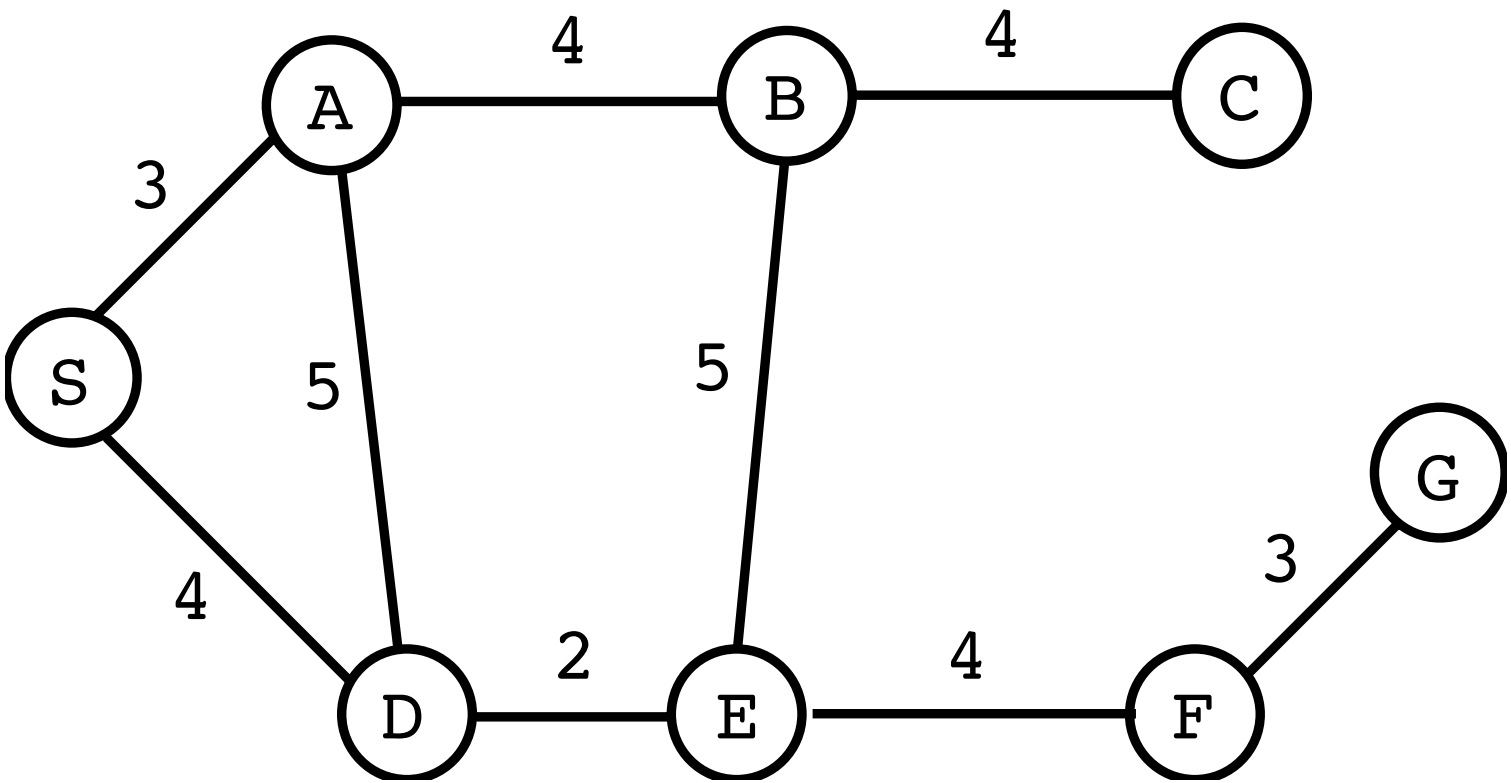
- La representación que se usa para caracterizar un problema es fundamental para su solución.
- Los requerimientos necesarios para una representación de un problema de búsqueda son:
  1. Una agenda que contiene conjuntos de soluciones potenciales
  2. Un conjunto de operaciones o reglas que nos modifican los símbolos en la agenda
  3. Una estrategia de búsqueda
- Por ejemplo, podemos jugar con un contrincante a seleccionar digitos en forma alternada, tratando de encontrar tres digitos que sumen 15. Si se usa un cuadro mágico y se juega gato el juego es trivial.

Representación

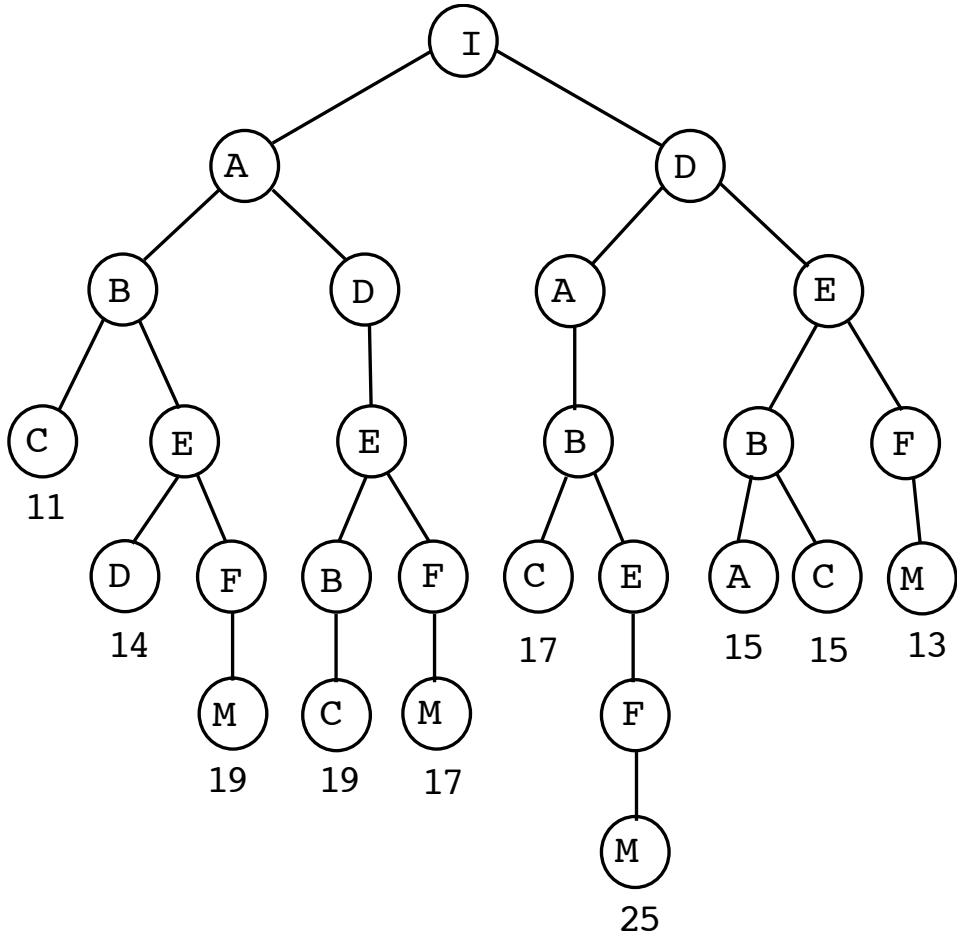
4	9	2
3	5	7
8	1	6

Figure: El cuadro mágico

Búsqueda Clásica



Búsqueda Clásica



Búsqueda Clásica

- Propiedades de algoritmos de búsqueda (heurísticas):
  1. *Completo*: un algoritmo se dice que es completo si encuentra una solución cuando ésta existe
  2. *Admisible*: Si garantiza regresar una solución óptima cuando ésta existe

Búsqueda ciega o sin información

- El orden en que la búsqueda se realiza no depende de la naturaleza de la solución buscada. La localización de la(s) meta(s) no altera el orden de expasión de los nodos.

Búsqueda ciega o sin información

Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta
    si el primer elemento es la meta
    entonces acaba
    si no elimina el primer elemento y
        añade sus sucesores al _____ de la agenda

Breadth–first search (búsqueda a lo ancho)

Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta
    si el primer elemento es la meta
    entonces acaba
    si no elimina el primer elemento y
        añade sus sucesores al *final* de la agenda

Breadth–first search (búsqueda a lo ancho)

- Breadth–first es completo (encuentra una solución si existe) y óptimo (encuentra la más corta) si el costo del camino es una función que no decrece con la profundidad del nodo.
- Pero requiere de mucha memoria.
- Si se tiene un factor de arborecencia de b y la meta está a profundidad d, entonces el máximo número de nodos epandidos antes de encontrar una solución es: $1 + b + b^2 + b^3 + \ldots + b^d$

Breadth–first search (búsqueda a lo ancho)

Profund.	Nodos	Tiempo	Memoria
0	1	1 miliseg.	100 bytes
2	111	.1 seg.	11 kilobytes
4	11,111	11 seg.	1 megabyte
6	10^6	18 min.	111 megabytes
8	10^8	31 hr.	11 gigabytes
10	10^{10}	128 días	1 terabyte
12	10^{12}	35 años	111 terabytes
14	10^{14}	3500 años	11,111 terabytes

Búsqueda de Costo Uniforme

- Una variante de breadth–first es expander todos los nodos por costos. Si el costo es igual a la profunidad se tiene el mismo algoritmo.
- La búsqueda de costo uniforme encuentra la solución más barata si el costo nunca decrece al aumentar los caminos.

$$costo(suc(n)) \geq costo(n)) \; \forall n$$

Depth–first o Búsqueda en Profundidad (LIFO)

Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta
    si el primer elemento es la meta
    entonces acaba
    si no elimina el primer elemento y
        añade sus sucesores al *frente* de la agenda

Depth–first o Búsqueda en Profundidad (LIFO)

- Depth–first necesita almacenar un solo camino de la raíz a una hoja junto con los "hermanos" no expandidos de cada nodo en el camino.
- Por lo que con un factor de arborecencia de b y profundidad máxima de m, su necesidad de almacenamiento es a los más bm.
- Depth–first no es ni completo ni óptimo y debe de evitarse en árboles de búsqueda de profunidad muy grande o infinita.

Backtracking

- Backtracking es una versión de depth–first que aplica el criterio LIFO para generar más que para expander. Esto es, sólo se genera un sucesor a la vez.
- Su gran ventaja es su ahorro en memoria. Su gran desventaja es el no poder incluir información para evaluar cual de los sucesores es el mejor.
- Algunas modificaciones de backtracking regresan al nodo que ocaciona que se llegue a un punto sin salida (*dependency directed backtracking*).
- Backtracking también puede servir para problemas de (semi–)optimización. Si mantenemos la solución más barata hasta el momento y usamos esa información para cortar caminos (asumiendo que el costo no decrece con la profunidad de la búsqueda).

Búsqueda con Profundidad Limitada (depth–limited)

- Para evitar caer en caminos de profunidad muy grande se impone un límite de profundidad máxima.
- Si se sabe la profundidad de alguna meta, el algoritmo puede ser completo, pero sigue sin ser óptimo.
- Cuando se tienen grafos altamente conectados hay que tener cuidado con el concepto de profundidad (una más que el padre menos profundo), se tienen que analizar todos los padres, lo cual implica mas memoria.
- En la práctica se puede tomar solo la profundidad del padre que lo generó haciendo una estrategia LIFO más “pura”.

Búsqueda de Profundidad Iterativa (*iterative* o *progressive deepening*)

- El problema con exigir un límite de profundidad, está precisamente en escoger un límite adecuado. Profundidad iterativa va aumentando gradualmente la profundidad hasta encontrar una meta. Es óptimo y completo como breadth–first pero requiere de poca memoria como depth–first.
- Por ejemplo, con un *branching factor* de 10 y con profundidad 5, los nodos expandidos serían $1 + 10 + 100 + \ldots + 100,000 = 111,111$, y con *iterative deepending* son: $(d+1)1 + (d)b + (d-1)db^2 + \ldots + 1 * b^d = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456 \approx 11\%$ más de nodos.
- Entre más grande sea el factor de arborecencia, menor el trabajo extra.

Búsqueda Bidireccional

- Aquí la idea es explorar al mismo tiempo del estado inicial hacia la meta que de la meta hacia el estado inicial hasta que se encuentren en un estado.
- Cuando el factor de arborecencia es igual en ambos sentidos, se pueden hacer grandes ahorros.

Puntos a Considerar

- Cuando los operadores son reversibles, los conjuntos de predecesores y sucesores son iguales, pero en algunos casos calcular los predecesores puede ser muy difícil.
- Si hay muchas posibles metas explícitas, en principio se podría hacer la búsqueda hacia atrás a partir de cada una de ellas. Sin embargo, a veces las metas son sólo implícitas (e.g., jaque mate).
- Se necesita un método eficiente para revisar si un nuevo nodo aparece en la otra mitad de la búsqueda.
- Se tiene que pensar que tipo de búsqueda hacer en cada mitad (no necesariamente la misma es la mejor).

Comparación de Algoritmos

Comparación de algoritmos. b = factor de arborescencia, d = profundidad de la solución, m = profundidad máxima, l = profundidad límite.

Criterio	Breadth first	Costo unif.	Depth first	Depth limited	Itera. deep.	Bidir.
Tiempo	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Espacio	b^d	b^d	$b \times m$	$b \times l$	$b \times d$	$b^{d/2}$
Optimo	si	si	no	no	si	si
Completo	si	si	no	si (si $l \geq d$)	si	si

Algoritmo Genérico con Información

Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta
    si el primer elemento es la meta
    entonces acaba
    si no elimina el primer elemento,
        añade sus sucesores a la agenda,
        ordena todos los elementos de la agenda

Hill–Climbing

Hill–climbing es una estrategia basada en optimización local.

Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta
si el primer elemento es la meta
entonces acaba
si no elimina el primer elemento,
añade sus sucesores a la agenda,
ordena todos los elementos de la agenda
selecciona el mejor y *elimina el resto*

Hill-Climbing

- Problemas obvios: máximos/mínimos locales, valles y riscos
- Para salir de minimos/máximos locales o tratar de evitarlos con hill–climbing a veces se empieza la búsqueda en varios puntos aleatorios (*random–restart hill–climbing*), salvando el mejor
- Dado su bajo costo computacional es la estrategia de búsqueda más utilizada en optimización y aprendizaje.

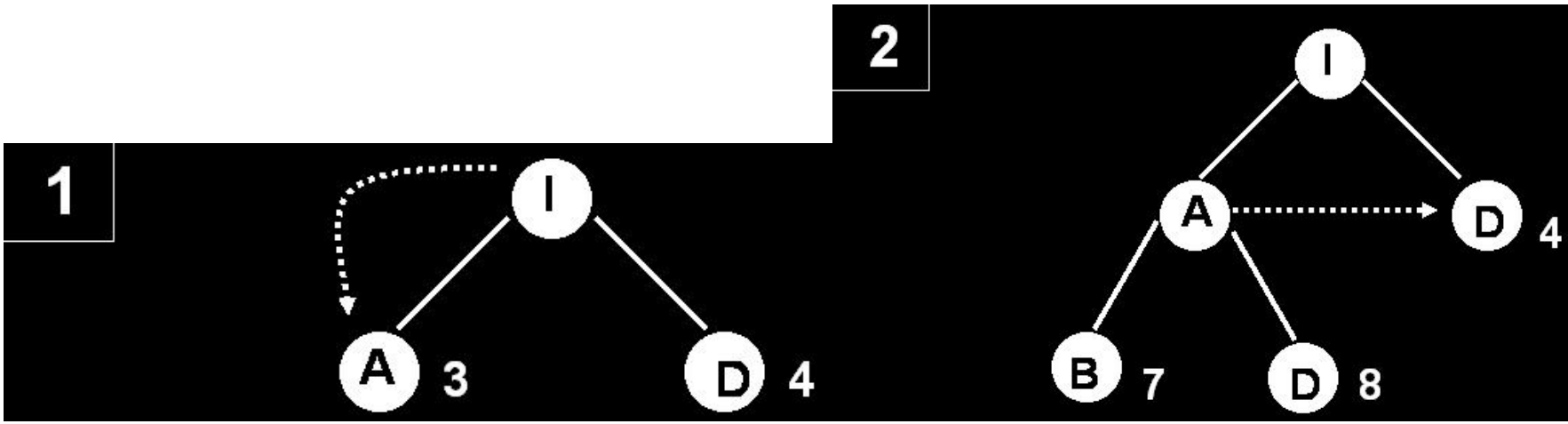
Best–First

Si el nodo que mejor evaluación recibe es el que se expande primero, entonces estamos haciendo “best–first”.

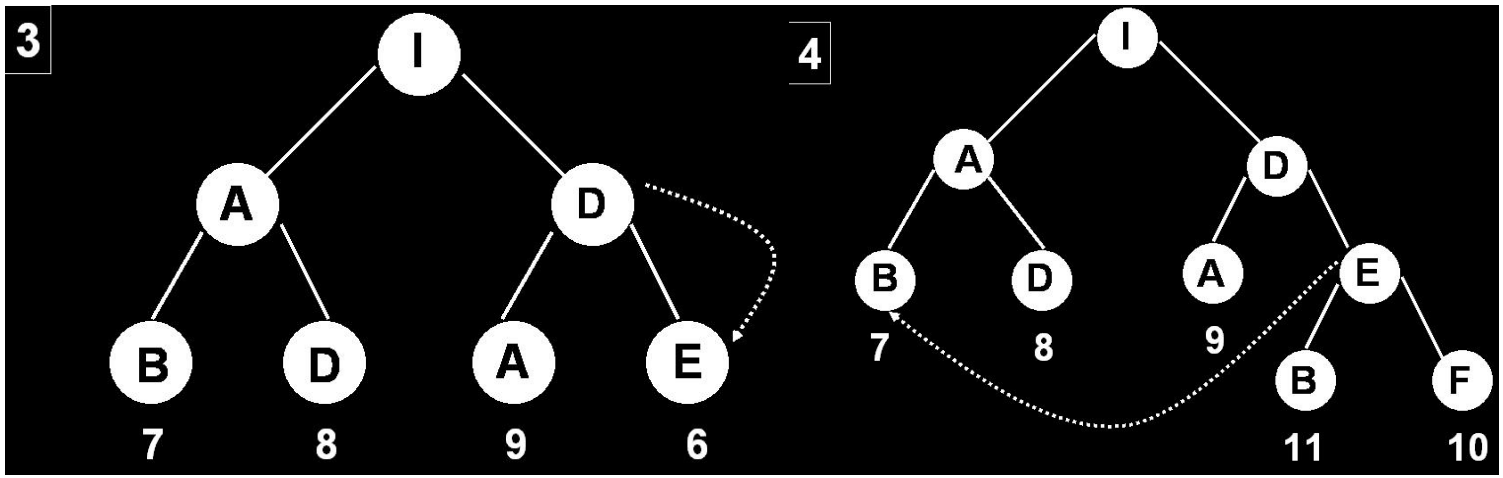
Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta
si el primer elemento es la meta
entonces acaba
si no elimina el primer elemento,
añade sus sucesores a la agenda,
ordena todos los elementos de la agenda

Best-First

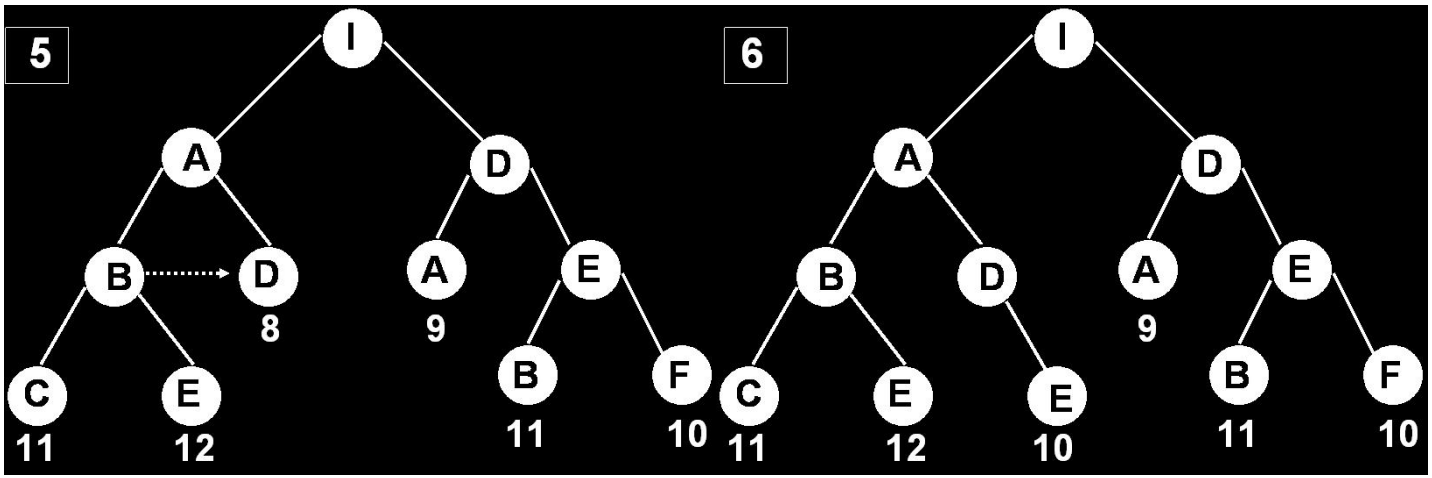
Más que estar expandiendo el “mejor”, se expande el que parece ser el mejor de acuerdo con nuestra función de evaluación.



Ejemplo



Ejemplo



Ejemplo

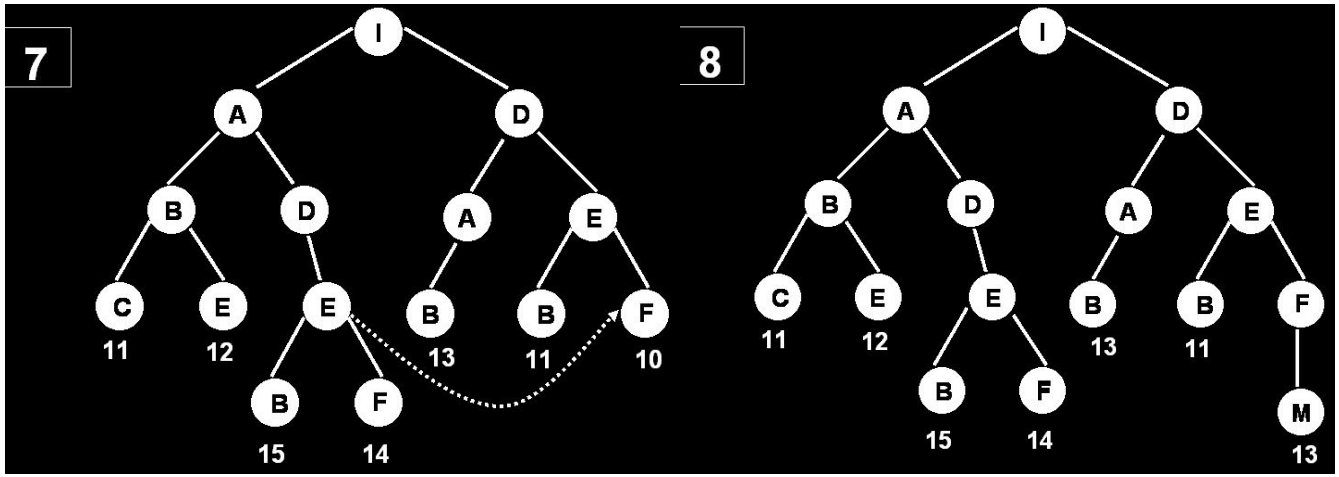


Figure: Búsqueda Best first

Usando Costo Estimado

- El usar el costo acumulado (búsqueda de costo uniforme) no necesariamente guía la búsqueda hacia la meta. Para ésto, se utiliza una estimación del costo del camino del estado hacia una meta ($h(n)$).
- Esta estrategia (minimizar el costo estimado para alcanzar una meta) a veces se llama una estrategia "greedy".
- La función de evaluación (h) puede ser lo que sea mientras $h(n) = 0$ si n es una meta.
- Debido a que guardan todos los nodos en memoria, su complejidad en espacio es igual a la del tiempo.

Híbridos

- Tres de las estrategias principales vistas: hill–climning, backtracking y best–first, se pueden ver como 3 puntos en un espectro continuo de estrategias, si las caracterizamos por:
  - Recuperación de caminos sin salida
  - Alcance de evaluación (número de alternativas consideradas)

Híbridos

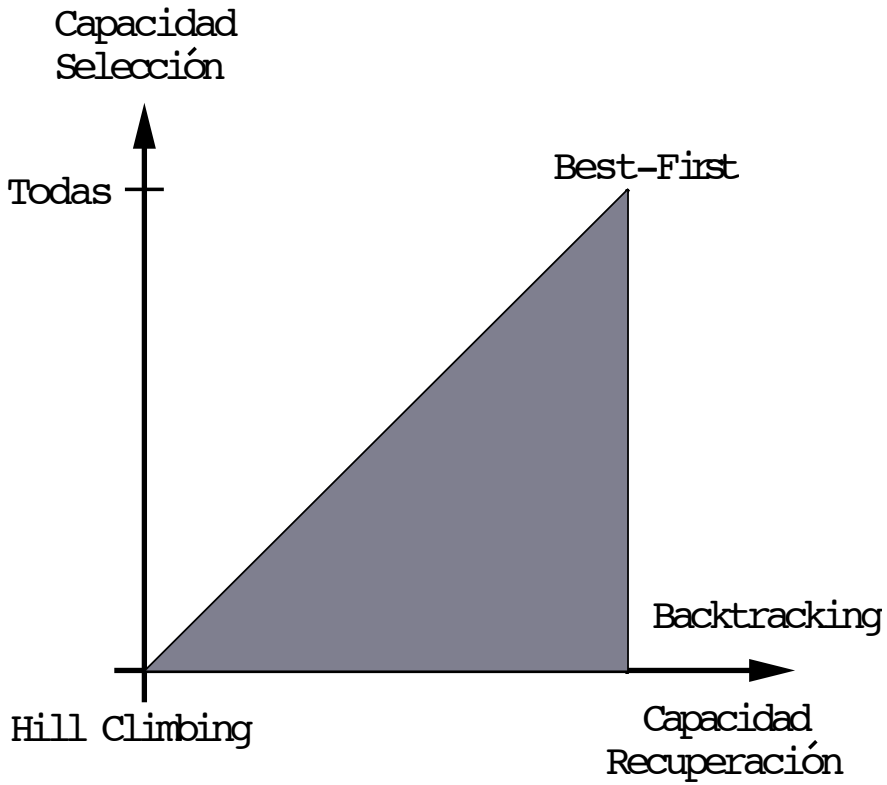


Figure: Algoritmos Híbridos

Híbridos

- Se puede hacer una combinación BF–BT. Se aplica BF al principio hasta agotar la memoria, seguido de BT. Si BT no encuentra una solución, se considera otro nodo.
- Otra posibilidad es usar BT hasta cierta profundidad y luego emplear BF. Si no se llega a una solución hacemos backtracking y buscamos en otra zona.
- Esta segunda opción es más difícil de controlar que la primera, pero aplicar BF al final tiene la ventaja que BF se comporta mejor entre más informado esté (lo cual es más probable que ocurra en la parte inferior del árbol)

Híbridos

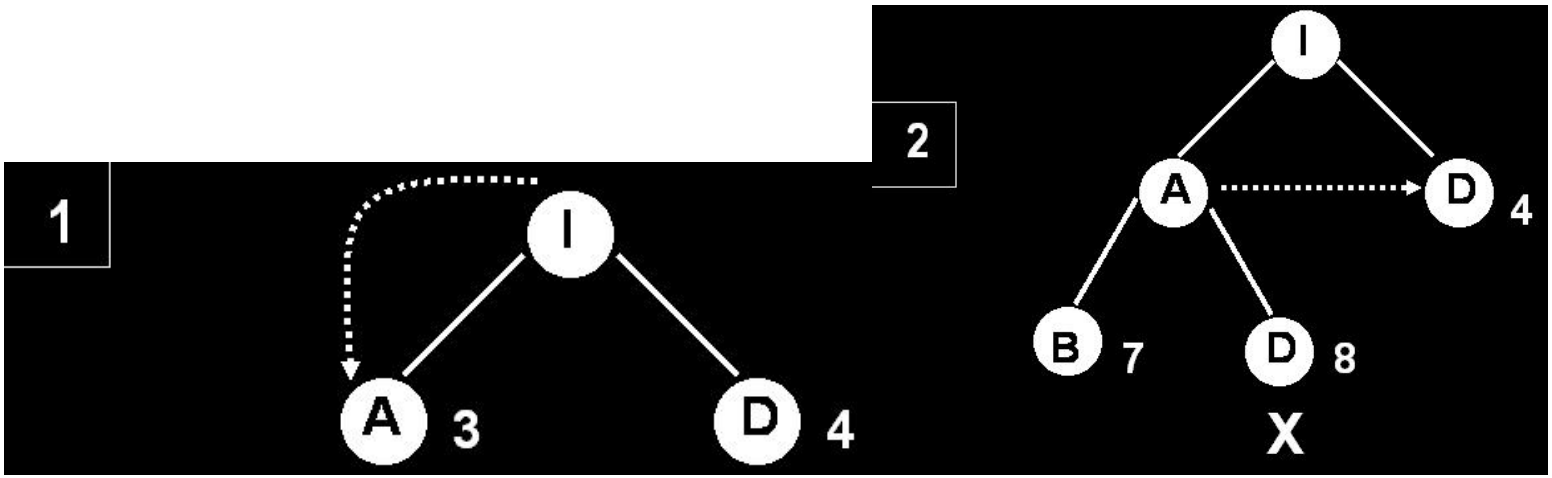
- Se puede tener una combinación de un BF local con un BT global.
- Se empieza con un BF con memoria limitada.
- Cuando se acaba la memoria, su lista de nodos en OPEN se toma como los sucesores directos del nodo de donde se partió (el nodo raíz la primera vez), los cuales se someten a un BT.
- Sin embargo, cada uno de ellos se expande usando un BF con memoria limitada.
- Si no se llega a la solución se hace *backtracking* a otro nodo.

Beam–Search

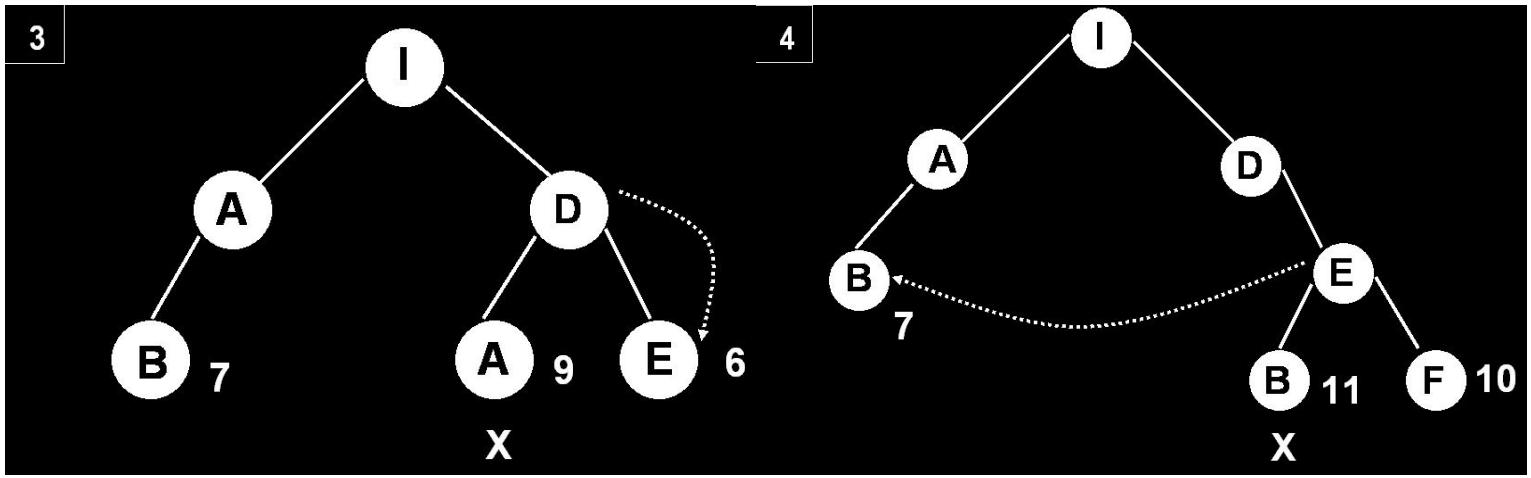
Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta
    si el primer elemento es la meta
    entonces acaba
    si no elimina el primer elemento,
        añade sus sucesores a la agenda,
        ordena todos los elementos de la agenda y
        selecciona los k mejores (elimina los demás)

Beam–Search

Si $k = 1$ es como hill-climbing, si k es tan grande para considerar todos los nodos de la agenda es como best-first.



Beam-Search



Ejemplo

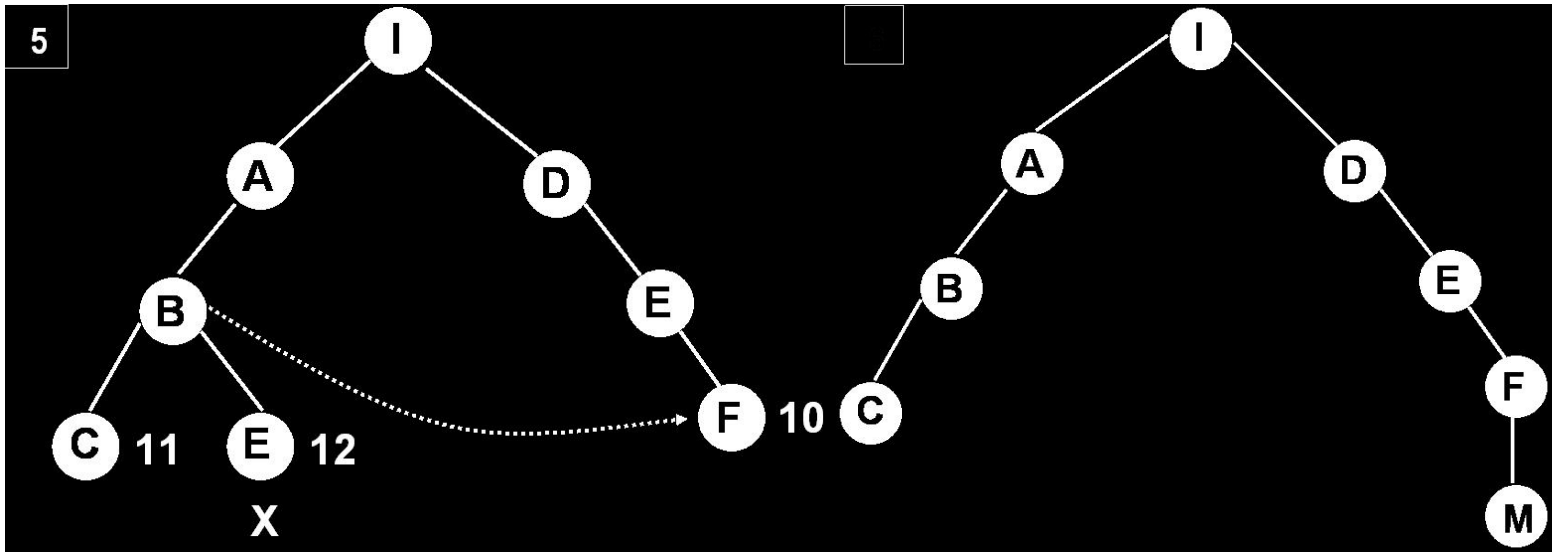


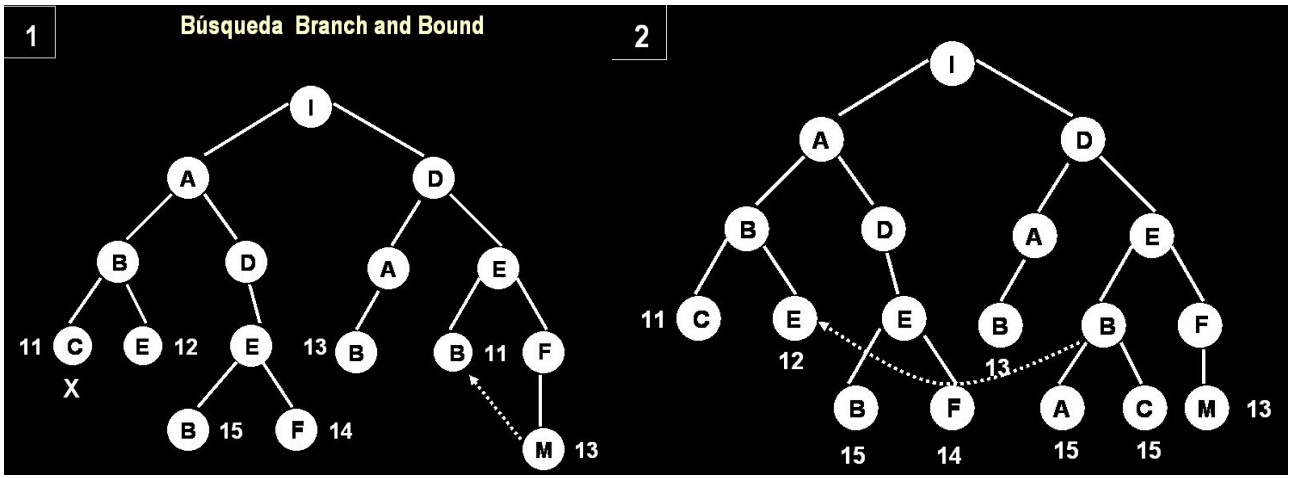
Figure: Búsqueda Beam search con $k = 2$.

Branch and Bound

Trabaja como best-first pero en cuanto se encuentra una solución sigue expandiendo los nodos de costos menores al encontrado

Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta y los demas nodos sean de costos mayores o iguales a la meta
si el primer elemento es la meta y los demas nodos son de menor o igual costo a la meta
entonces acaba
si no elimina el primer elemento y añade sus sucesores a la agenda
ordena todos los elementos de la agenda

Ejemplo



Ejemplo

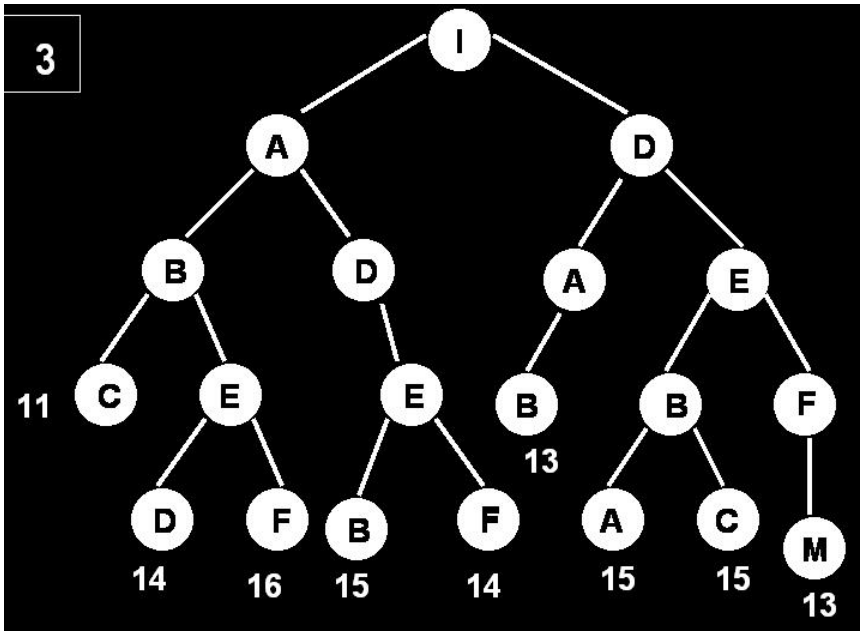


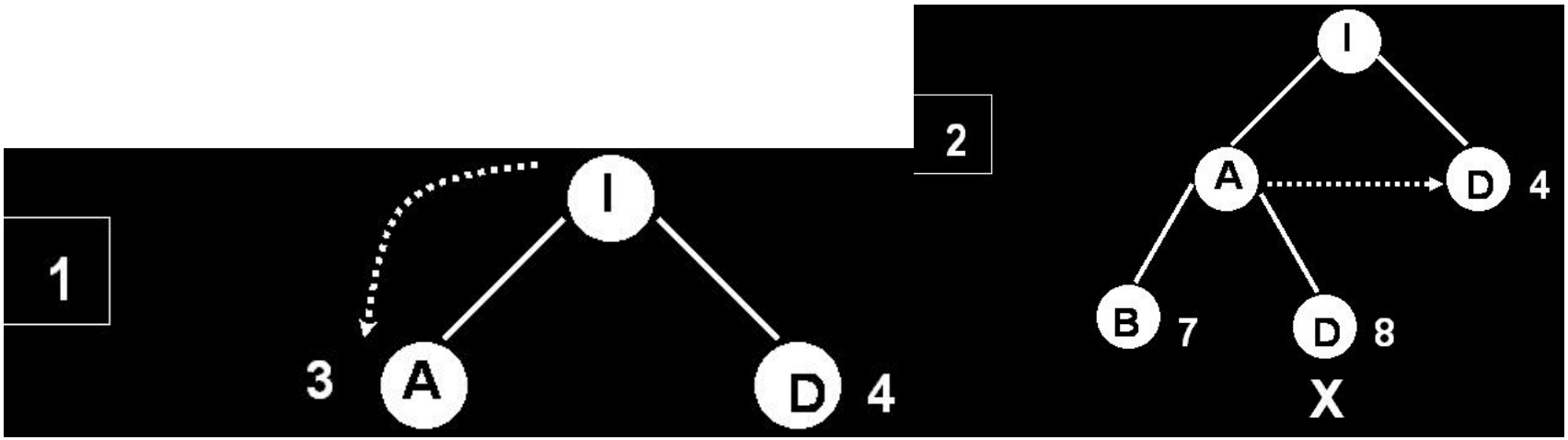
Figure: Búsqueda Branch and Bound

Dynamic Programming

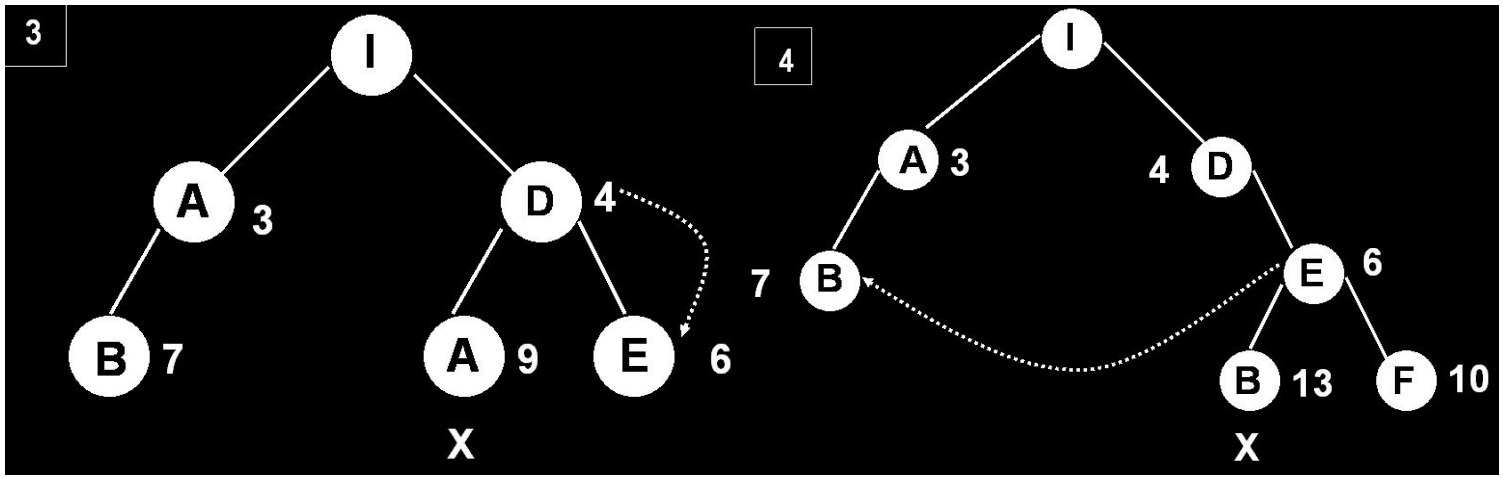
- Idea: no explorar caminos a los que yá llegamos por caminos más cortos/baratos.
- El algoritmo es igual sólo hay que añadir la condición:

  elimina todos los caminos que lleguen al mismo nodo excepto el de menor costo

Ejemplo



Ejemplo



Ejemplo

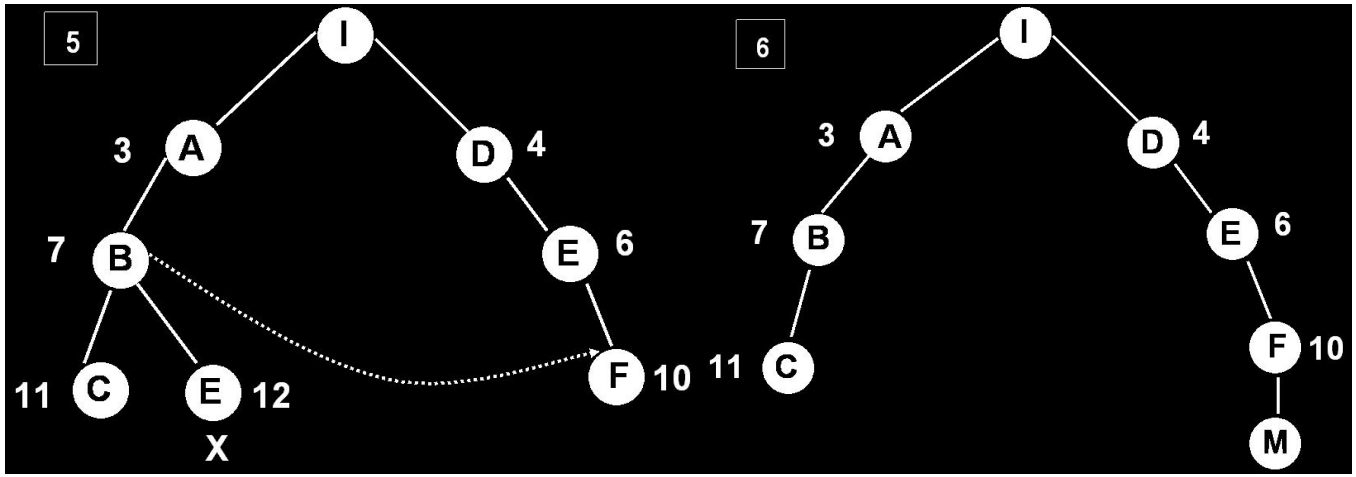


Figure: Búsqueda con programación dinámica

A*

- Búsqueda "greedy" minimiza $h(n)$ reduciendo la búsqueda, pero no es ni óptima ni completa. Búsqueda de costo uniforme minimiza el costo acumulado $g(n)$ y es completa y óptima, pero puede ser muy ineficiente.
- Se pueden combinar las dos sumandolas, usando para cada nodo la siguiente heurística: $f(n) = g(n) + h(n)$.
- La estrategia se puede probar que es completa y óptima si $h(n)$ nunca sobre–estima el costo. Tal función se le conoce como una heurística *admisible*.
- Breadth–first se puede ver como un caso especial de A* si $h = 0$ y $c(n, n') = 1$ para todos sus sucesores (el costo de ir de un nodo padre a un nodo hijo).
- Si h es admisible, $f(n)$ nunca hace sobrestimaciones.

Algoritmo A*

Crea una agenda de un elemento (el nodo raíz)
hasta que la agenda esté vacía o se alcance la meta y los demas nodos sean de costos mayores o iguales a la meta
si el primer elemento es la meta y los demas nodos son de menor o igual costo a la meta
entonces acaba
si no elimina el primer elemento y añade sus sucesores a la agenda
ordena todos los elementos de la agenda de acuerdo al costo acumulado más las subestimaciones de los que falta
elimina todos los caminos que lleguen al mismo nodo, excepto el de menor costo

Ejemplo

Ejemplo

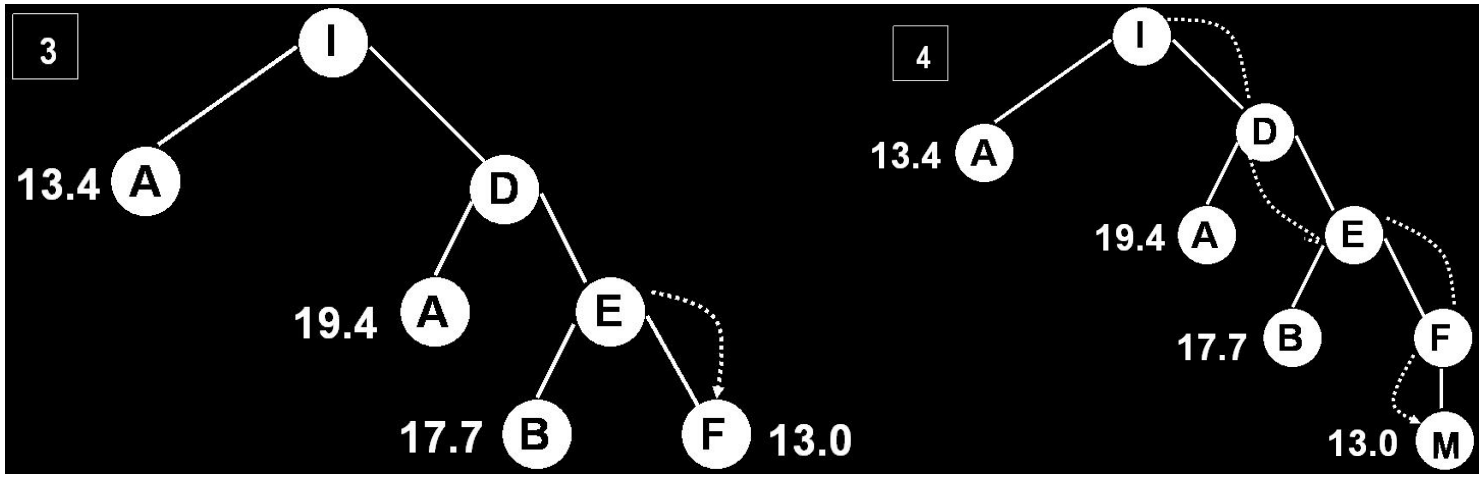


Figure: Búsqueda A*

Comparación

Resultados promedios de 100 instancias del *8-puzzle*.

	Costo de búsqueda			Arborecencia efetiva		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Extensiones

- Pesar la combinación de g (lo que llevo) y de h (lo que me falta)
- Por ejemplo:

$$f(n) = (1 - w)g(n) + wh(n)$$

$w = o \Rightarrow$ búsqueda de costo uniforme
$w = 1/2 \Rightarrow$ A*
$w = 1 \Rightarrow$ BF

- Usar un peso dinámico que se ajusta con la profundidad, e.g.:

$$f(n) = g(n) + h(n) + \epsilon[1 - \frac{d(n)}{N}]h(n)$$

donde $d(n)$ es la profundidad del nodo n y N la profundidad anticipada del nodo meta

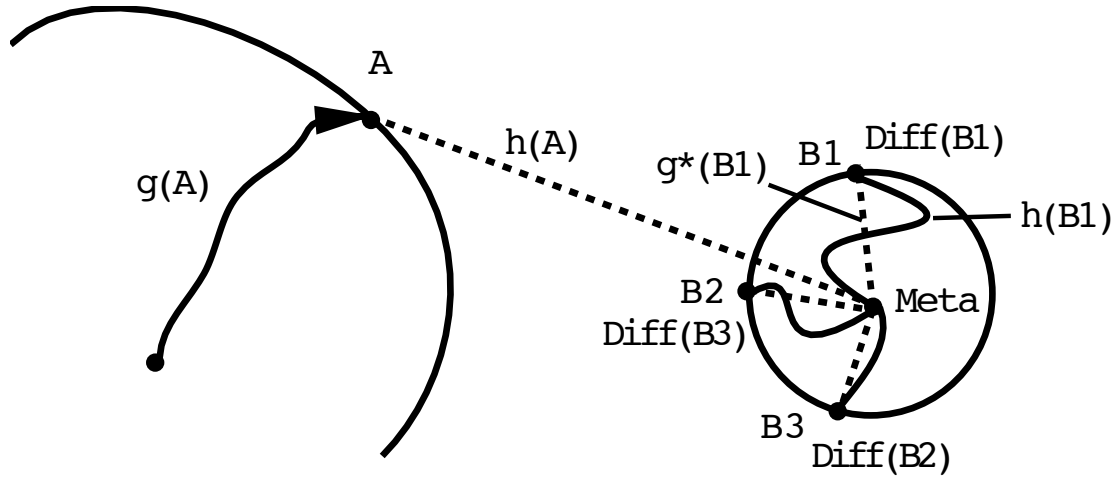
- Se pueden usar otras posibles medidas no aditivas para h y g: e.g., multiplicativa.

Extensiones

- IDA*, extensión natural de *iteractive deepening* con límite de costo iterativo
- Cuando se tienen muchas posibilidades más o menos todas del mismo costo, lo que se puede hacer es agruparlas todas en un mismo rango y sólo expander la mejor de acuerdo a cierta heurística
- Es como A*, sólo que ahora se tienen dos listas: OPEN (como antes) y FOCAL $\subset$ OPEN, en donde se tienen todos los nodos que no se desvían de la mejor opción en más de ϵ.
- MA* al llenar su memoria elimina el nodo de mayor costo de su agenda y propaga su costo a su nodo padre.

Extensiones

- Las evaluaciones en las heurísticas son estáticas. Una forma de mejorarlas es actualizarlas dinámicamente conforme se obtiene más informacón
- Algunas variantes incluyen (aunque existen otras):
  1. *Add method*
  2. *Max method*
  3. *Back-up method*
  4. *Front-to-front method*



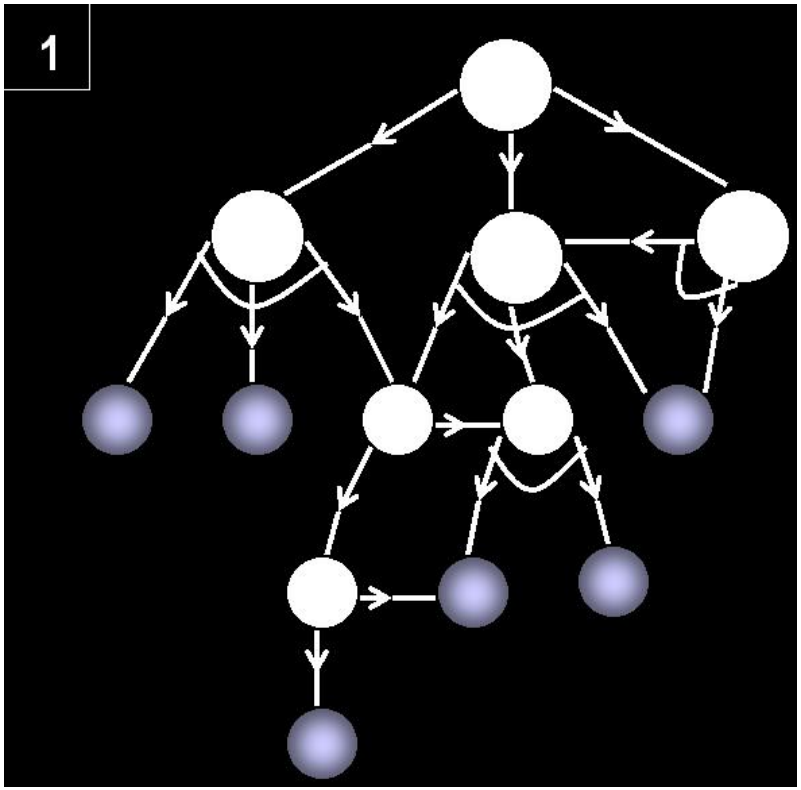
e.g., *Add*

Búsqueda en grafos AND-OR

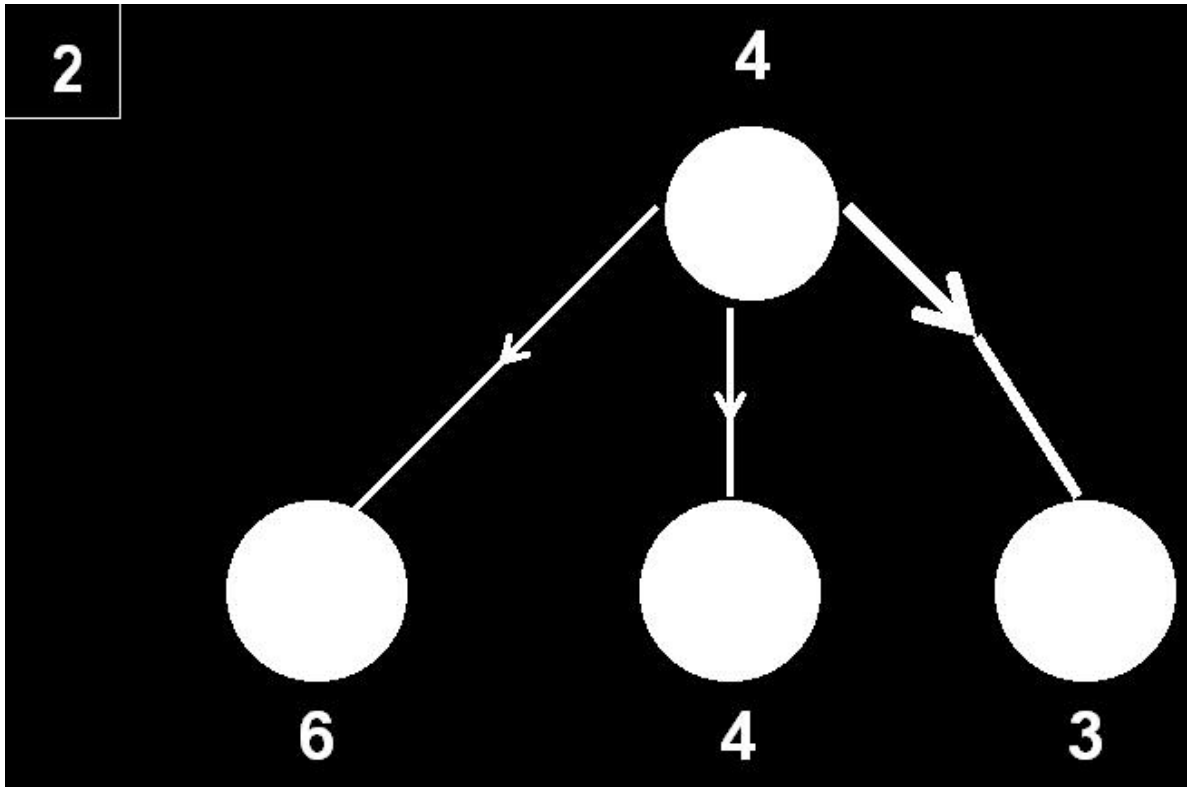
- Todas las estrategias vistas tienen su equivalente para árboles AND-OR
- Por ejemplo en depth-first:

  IF algún nodo AND falla
  realiza backtracking hasta el último nodo OR
  IF algún nodo OR falla
  realiza backtracking hasta el nodo inmediato anterior

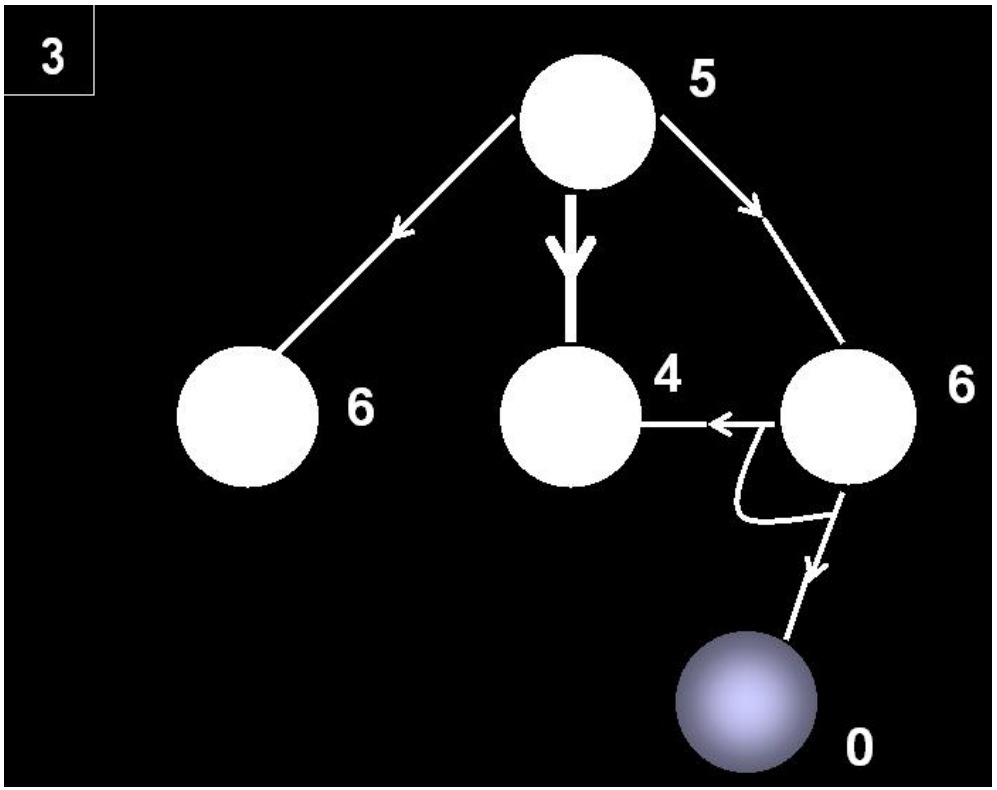
AO*



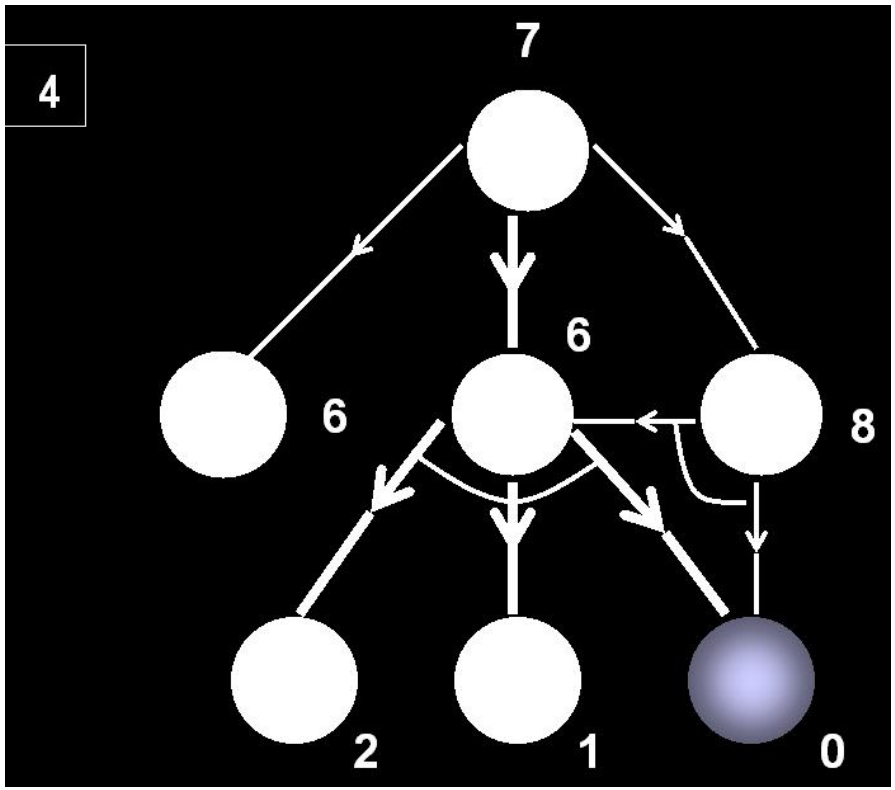
AO*



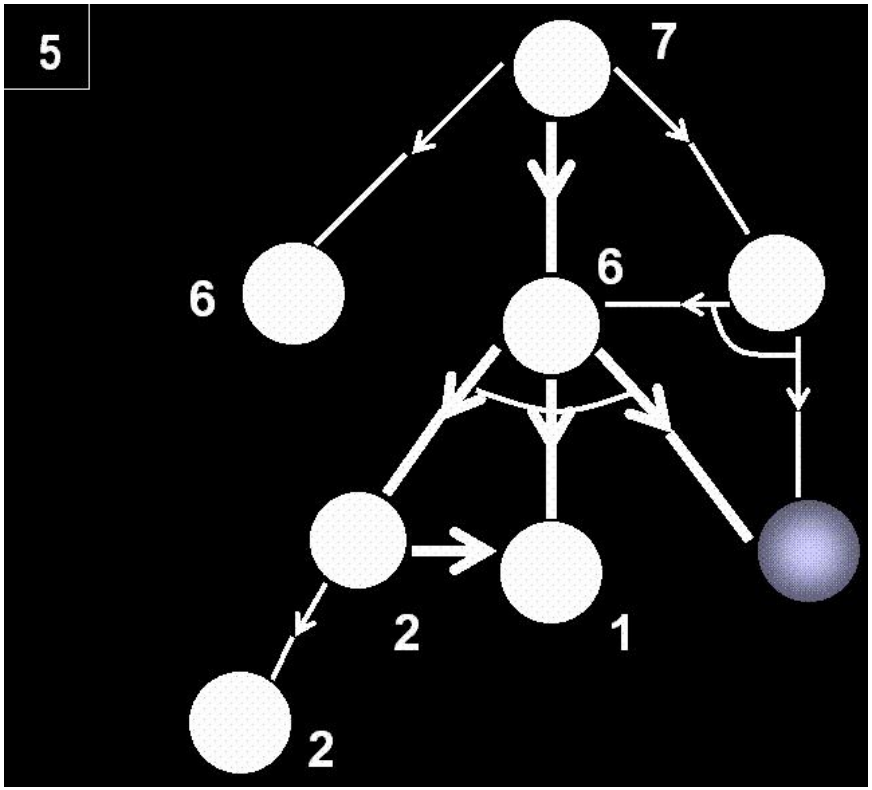
AO*



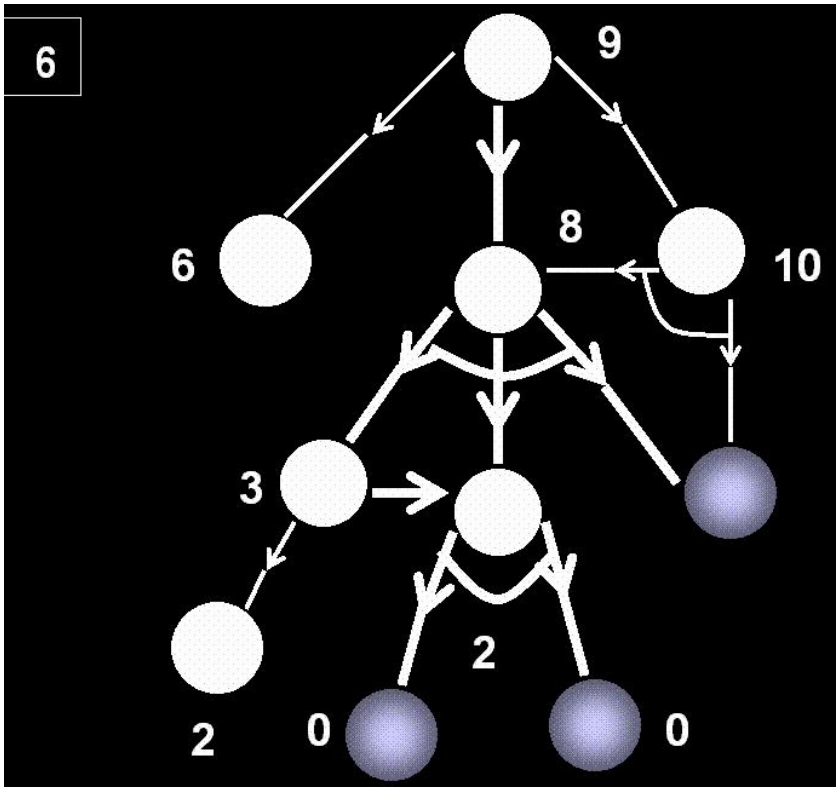
AO*



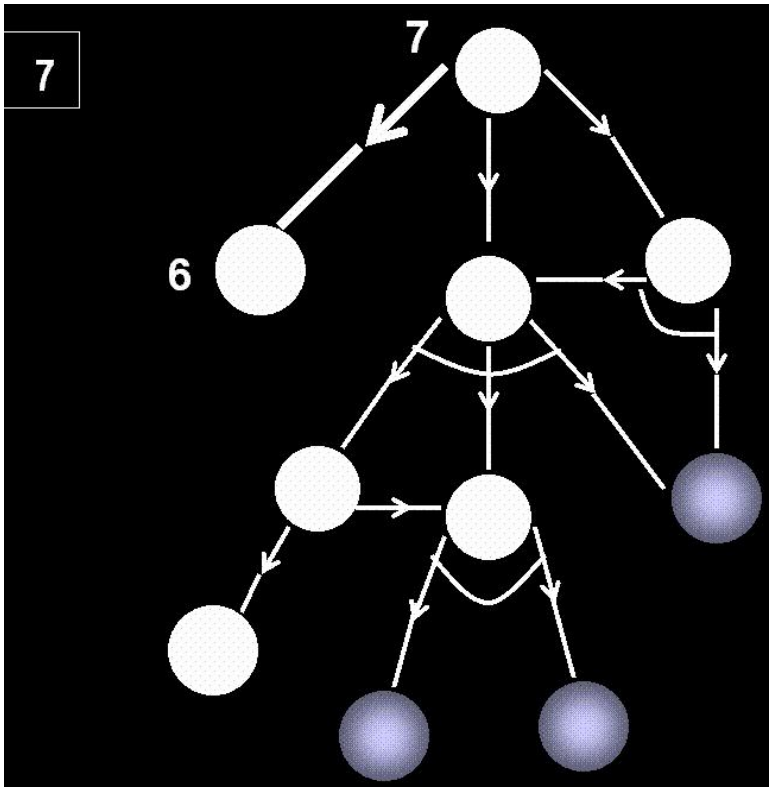
AO*



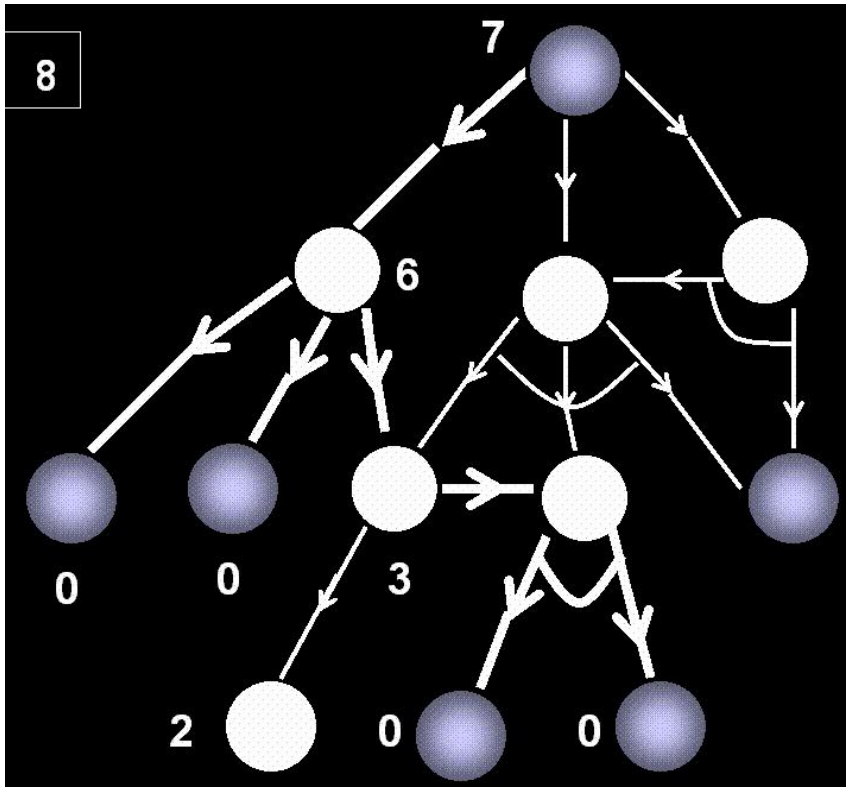
AO*



AO*



AO*



Búsqueda en Prolog


```
busca(NodoI,NodoF) :-
    busca_aux([NodoI],NodoF).

busca_aux([NodoF|_],NodoF).
busca_aux(Agenda,NodoF) :-
    nva_agenda(Agenda,NAgenda),
    busca_aux(NAgenda,NodoF).
```

Búsqueda en Prolog


```
% depth-first
nva_agenda([N1|Agenda],NAgenda) :-
    expande(N1,Nodos),
    append(Nodos,Agenda,NAgenda).

% breadth-first
nva_agenda([N1|Agenda],NAgenda) :-
    expande(N1,Nodos),
    append(Agenda,Nodos,NAgenda).
```

Búsqueda en Prolog


```
% best-first
nva_agenda([N1|Agenda],NAgenda) :-
    expande(N1,Nodos),
    append(Nodos,Agenda,AgendaInt),
    sort(AgendaInt,NAgenda).

% hill-climbing
nva_agenda([N1|Agenda],[Mejor]) :-
    expande(N1,Nodos),
    append(Nodos,Agenda,AgendaInt),
    sort(AgendaInt,[Mejor|_]).
```

Búsqueda en Prolog


```
% beam search
nva_agenda(Beam,[N1|Agenda],NAgenda) :-
    expande(N1,Nodos),
    append(Nodos,Agenda,AgendaInt),
    sort(AgendaInt,AgendaOrd),
    nthelems(Beam,AgendaOrd,NAgenda).
```

Búsqueda en Lisp


```
(defun busca(nodoI,nodoF)
    (busca_aux (list nodoI) nodoF))

(defun busca_aux(agenda nodoF)
    (cond ((null agenda) nil)
          ((equal (car agenda) nodoF))
          (t (busca_aux(nva_agenda (car agenda)
               (cdr agenda)) nodoF))))
```

Búsqueda en Lisp


```
; breadth-dirst
(defun nva_agenda(nodo agenda)
     (append (expande nodo) agenda))

; depth search
(defun nva_agenda(nodo agenda)
     (append agenda (expande nodo)))

; best-first
(defun nva_agenda(nodo agenda)
     (sort (append (expande nodo) agenda)))
```

Búsqueda en Lisp


```
; hill-climbing
(defun nva_agenda(nodo agenda)
     (list (car (sort (append (expande nodo) agenda)))))

; beam search
(defun nva_agenda(beam nodo agenda)
     (nthelems beam (sort (append agenda (expande nodo)))))
```

Cómo Inventar Heurísticas

- Muchas veces el costo de una solución exacta para un problema con menos restricciones o simplificado (*relaxed*) es una buena heurística para el problema original.
- Por ejemplo, podemos usar la notación de STRIPS para representar movimientos

Cómo Inventar Heurísticas

La accción de mover un cuadro en el 8-puzzle usando notación de Strips.

MUEVE(CuadroX,LugarY,LugarZ):
- Precondiciones:
  - EN(CuadroX,LugarY),
  - LIBRE(CuadroZ),
  - ADY(CuadroY,CuadroZ)
- Añade:
  - EN(CuadroX, LugarZ),
  - LIBRE(LugarY)
- Elimina:
  - EN(CuadroX,LugarY),
  - LIBRE(LugarZ)

Cómo Inventar Heurísticas

- Si eliminamos las condiciones: LIBRE(CuadroZ), ADY(CuadroY,CuadroZ) nos queda un problema en donde cada cuadro se puede mover a cualquier otro. El número requerido de pasos para resolver este problema es igual al número de cuadros que no están en su lugar (h_1).
- Si sólo eliminamos LIBRE(CuadroZ), corresponde a intercambiar dos cuadros y nos dá la heurística h_2.
- Si eliminamos sólo ADY(CuadroY,CuadroZ), nos dá otra heurística que se introdujo 13 años después de las otras dos.

Cómo Inventar Heurísticas

- Si queremos añadir más posibilidades, podemos expresar relaciones, tales como *ADJ* en más simples, eg., VECINOS y MISMA–LINEA.
- Eliminando una de las dos, dá nuevas heurísticas.
- Hay que tener cuidado, a veces el problema resultante puede ser más difícil de resolver.
- Si tenemos un conjunto de heurísticas admisibles lo mejor es hacer: $h(n) = max(h_1(n), \ldots, h_m(n))$ con lo cual h domina a todas.
- Otra forma de inventar heurísticas es usando información estadística. Si nuestra heurística nos da un cierto valor diferente podemos usar ese otro valor. Esto deja de garantizar la admisibilidad, pero puede dar en promedio muy buenos resultados.

MACRO–Operadores

- Meta = encontrar una secuencia de operadores que mapea el estado inicial a alguno de los estados finales.
- En muchos casos, algunas de las secuencias de operadores se repiten en problemas diferentes.
- Idea principal: juntar estas secuencias de operadores en macro operadores, “chunks”, etc.
- Considerar a los macros como operadores primarios para aumentar la velocidad de solución en problemas similares.

MACRO–Operadores

Los macros:

- Reducen la profundidad del árbol de búsqueda al considerar secuencias largas de operadores como una sola.
- Aumentan el *branching factor*.
- En el 8-puzzle, las 181,440 posiciones posibles, se pueden resolver sin búsqueda usando una tabla de 35 macros.

MACRO–Operadores

- La tabla de macro operadores para el *8-puzzle* se muestra en la siguiente tabla
- Cada operador es una secuencia de movimientos: up (U), down (D), left (L), right (R) de un cuadro.
- Para usarla se localiza el espacio (cuadro 0) y dependiendo de su posición se realizan los movimientos indicados en la columna de cuadro 0. Luego se sigue con el cuadro 1 y asi sucesivamente.
- Estos macros llegan a la posición final mostrada en la figura 2.

MACRO–Operadores

Macro operadores para el *8-puzzle* (P = posición 0).

P	Cuadros			
	0	1	2	3
0				
1	UL			
2	U	RDLU		
3	UR	DLURRDLU	DLUR	
4	R	LDRURDLU	LDRU	RDLLURDRUL
5	DR	ULDRURD-LDRUL	LURDLDRU	LDRULURDDLUR
6	D	URDLDRUL	ULDDRU	URDDLULDRRUL
7	DL	RULDDRUL	DRUULDRDLU	RULDRDLUL-DRRUL
8	L	DRUL	RULLDDRU	RDLULDRRUL

MACRO–Operadores

Tabla de macro operadores para el *8-puzzle* (cont.)

Pos	Cuadros		
	4	5	6
5	LURD		
6	ULDR	RDLLUURDLDRRUL	
7	URDLULDR	ULDRURDLLURD	URDL
8	RULLDR	ULDRRULDLURD	RULD

MACRO–Operadores

- En el cubo de Rubik ($4 * 10^{19}$) se usan 238 macros.
- La descomposición de problemas en sub–metas juega un papel fundamental en la creación de macros.
- El beneficio de los macros decae al incrementar su número (i.e., al resolver más problemas) y es necesario usar filtros de *aplicabilidad*.

MACRO–Operadores

- Al aplicar los filtros hay que considerar que los macros no son unidades totalmente independientes.
- Se deben de considerar cuándo y bajo qué condiciones crear/eliminar macros
- En general los macros aprendidos y su filtrado dependen de la función de evaluación (heurística) que se usa durante la búsqueda de la solución del problema.

MACRO–Operadores

En general:

- Una buena función de evaluación requiere de un filtro *agresivo*.
- Con una mala función de evaluación los macors son indispensables y el filtro puede ser más *laxo*.
- Con una muy buena función de evaluación los macros no sirven.
- Se pueden utilizar en aprendizaje incremental.

Tarea

Partiendo del siguiente estado inicial del 8-puzzle y con el siguiente estado final, realiza las siguientes estrategias de búsqueda:

- Breadth-first
- Best-first
- Hill-climbing

4	2	1
7	3	5
	8	6

$\Rightarrow$

1	2	3
4	5	6
7	8	

Juegos

La habilidad de jugar es considerada como una distinción de inteligencia.
Características:

- Fácil de crear situaciones complicadas con reglas sencillas.
- Se pueden probar contra humanos en donde existen escalas.
- Son adictivos.

Juegos

- Casi todos: 2 jugadores + *información perfecta*, aunque existen otros, por ejemplo: barajas, backgammon, etc.
- Con información perfecta: las reglas del juego determinan los posibles movimientos.
- A diferencia de búsqueda, el oponente introduce incertidumbre porque no sabemos qué va a tirar, lo cuál se asemeja más a problemas reales.

Juegos

En un juego tenemos:

- Posición inicial.
- Conjunto de operadores (definen movidas legales).
- Estado terminal.
- Función de utilidad, e.g., gana, pierde, empata (pero puede haber más, por ejemplo en backgammon).

Juegos

- Los árboles de juegos tienen correspondencia con árboles AND–OR (mis tiradas son OR, las del oponente son AND).
- Se require de una estrategia que garantice llegar a estados terminales ganadores independientemente de lo que haga el oponente.
- Los juegos, y el ajedrez en particular, han sido objeto de estudio por muchos años por los investigadores de Inteligencia Artificial.

MimiMax

- Shannon lo sugirió originalmente (50), basado en teoría de juegos de von Neumann y O. Morgenstern, aunque Turing presentó el primer programa (51).
- Idea: Maximizar mis tiradas considerando que el oponente va a minimizar.
- Para decidir qué jugada hacer, el árbol se divide por niveles:
  - *Max*: El primer jugador (nivel) y todas las posiciones (niveles) donde juega.
  - *Min*: El oponente y todas las posiciones en donde juega

Minimax

- Las hojas se etiquetan con *gana, pierde*, *empata* desde el punto de vista de *max*.
- Si podemos etiquetar todas las hojas, podemos etiquetar todo el árbol.
- La siguiente función calcula lo mejor que *max* puede esperar desde la posición *J* si juega de manera óptima contra un oponente perfecto.

Minimax

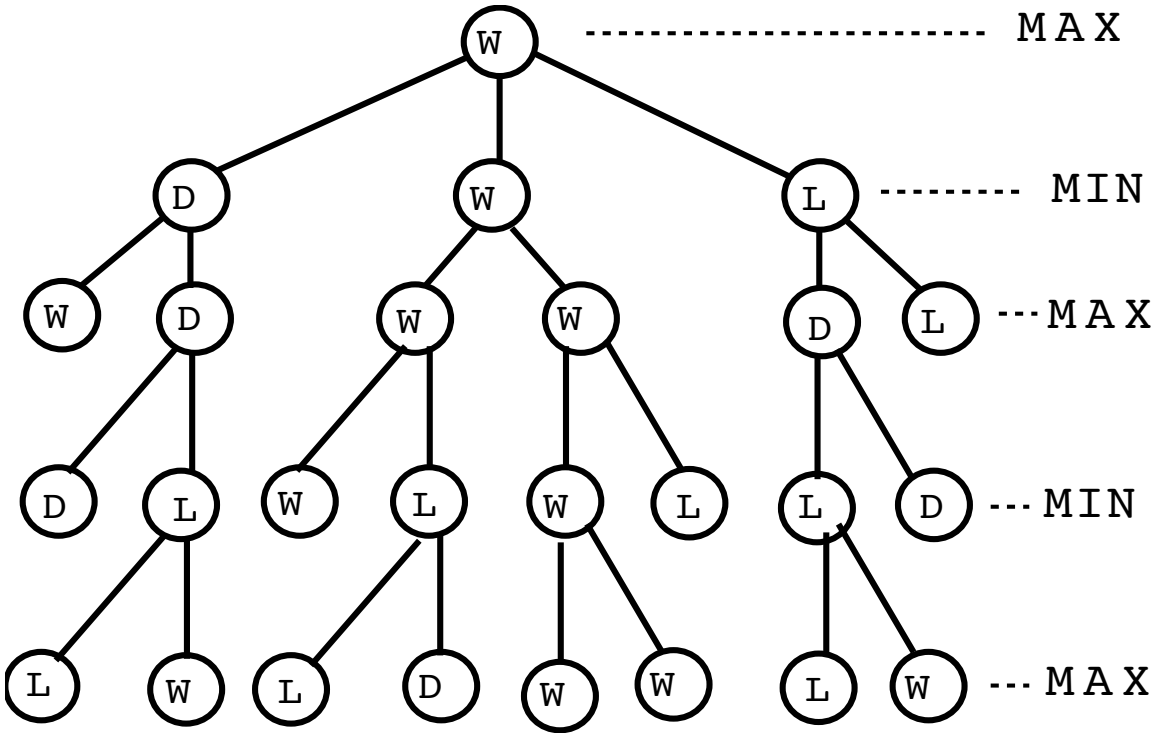
Si J es nodo *max* no–terminal:

$$etiq(J) = \begin{cases} gana & \text{si algún sucesor de } J \text{ es } gana \\ pierde & \text{si todos los sucesores son } pierde \\ empata & \text{si alguno } empata \text{ y ninguno } gana \end{cases}$$

Si J es nodo *min* no–terminal:

$$etiq(J) = \begin{cases} gana & \text{si todos los sucesores de } J \text{ son } gana \\ pierde & \text{si algún sucesor de } J \text{ es } pierde \\ empata & \text{si alguno } empata \text{ y ninguno } pierde \end{cases}$$

Minimax



Minimax

- Una estrategia para un jugador es un árbol considerando un camino para cada nodo del jugador y todos los posibles para el oponente.
- Desde el punto de vista de *max*:
  - *min* tira hacia la estrategia menos favorable: $min(estrategias)$
  - *max* trata de maximizar ésto: $max(min(estrategias))$
- Si cambiamos los roles (tira *min*): $min(max(estrategias))$.
- *Minimax* propaga suponiendo que las estimaciones son correctas.
- Si la estimación es burda, la propagación también va a ser burda.

Negmax

Negmax: Igual a *minimax*, pero la función de evaluación es simétrica, por lo que:

$$etiq(J) = \begin{cases} gana & \text{si algún sucesor de } J \text{ es } pierde \\ pierde & \text{si todos los sucesores de } J \text{ son } gana \\ empata & \text{cualquier otra situación} \end{cases}$$

$$status(J) = max_{J'}\{-status(J') \mid J' \text{ es hijo de } J\}$$

Lo que nos interesa son estrategias de juegos ganadoras sin importar lo que haga el oponente (*min*).

Tamaño de Búsqueda

- Una de las motivaciones principales de investigación en juegos es que son difíciles de resolver.
- Ajedrez: *branching factor* ≈ 35, número promedio de jugadas por jugador ≈ 50, por lo que hay $\approx 35^{100}$ o 10^{120} hojas o nodos terminales (aunque *sólo* hay 10^{40} posiciones diferentes legales).
- Algunos autores sugieren 80 jugadas promedio por jugador y un árbol de búsqueda de 10^{50}

Tamaño de Búsqueda

- Como no se puede evaluar/explorar todo el árbol, exploramos hasta cierta profundidad y usamos heurísticas (una función de evaluación *estática*).
- Suposición: Evaluando después de cierta exploración es mejor que evaluar sin exploración, bajo el supuesto que el mérito de una situación se clarifica al ir explorando.

Juegos

- La calidad del programa depende fundamentalmente de la función de evaluación.
- Tiene que ser:
  1. Congruente con la función de utilidad de los nodos terminales,
  2. Rápida de evaluar
  3. Reflejar las posibilidades actuales de ganar.
- Una valor de la función de evaluación en realidad cubre muchas posibles posiciones.
- Si fuera perfecta, no tendríamos que hacer búsqueda.

Juegos

- El algoritmo *minimax* usa dos heurísticas: (i) cálculo de la función de evaluación y (ii) propagar valores hacia atrás suponiendo que los valores en los nodos de frontera son correctos.
- Si las estimaciones son burdas, también son los resultados.
- Muchas de las funciones de evaluación suponen independencia entre los factores y las expresan como funciones lineales con pesos: $w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$
- Para ésto, hay que encontrar los pesos y decidir qué factores a considerar.

Juegos

Algunas estrategias de evaluación (tomadas de Deep–Blue antes Deep–Thought):

- Material: peón = 1, caballos y alfiles = 3, torre = 5, y reina = 9. Puede variar dependiendo de la situación (e.g., el mantener un par de alfiles al final del juego).
- Posición: Al principio se pensó que el control del centro era lo primordial. Aunque si es muy importante no es lo único. Una forma fácil de imaginarse "posición" es contanto el número de cuadros que se pueden atacar en forma segura. Entre más cuadros se tengan, se tiene más control.
- Defensa: Los aspectos defensivos de la posición tienen que ver con la seguridad del rey.
- Tempo: define la "carrera" por el control del tablero.

Minimax

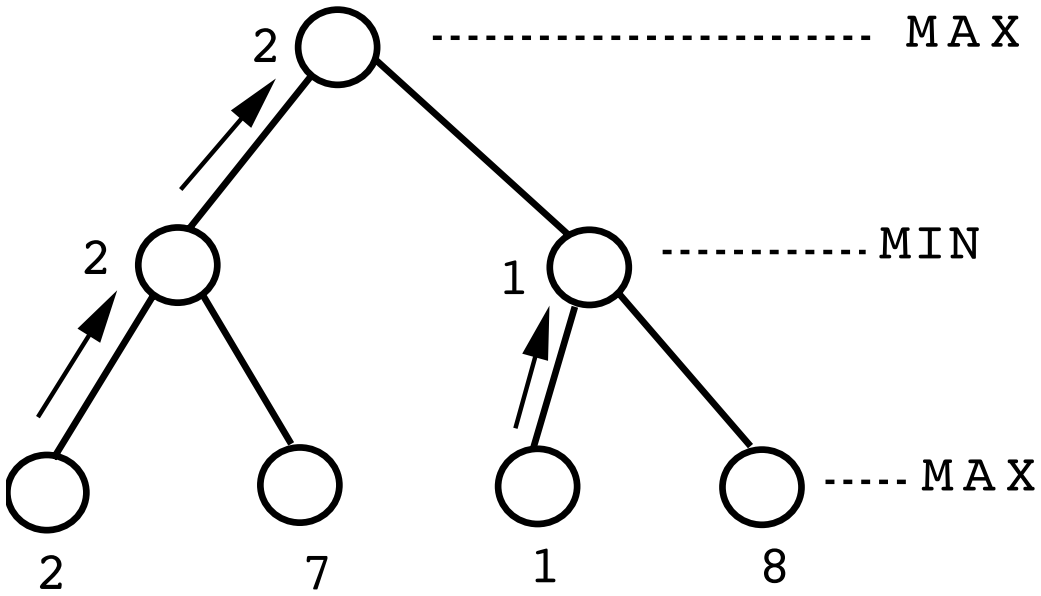
- La mayoría de los programas usan variantes de *minimax*: (i) generar árbol hasta cierta profundidad y (ii) evaluar posiciones en nodos de frontera.
- El esfuerzo (dada una función de evaluación) es proporcional al número de nodos en la frontera que son evaluados.

Juegos

Para determinar el valor *minimax* de J: $V(J)$

si J es terminal, $V(J) \leftarrow ev(J)$
sino genera los sucesores de J : $J_1, J_2, \ldots, J_n$
evalua $V(J_1), V(J_2), \ldots, V(J_n)$ de izquierda a derecha
si J es nodo *max*, $V(J) \leftarrow max[V(J_1), \ldots, V(J_n)]$
si J es nodo *min*, $V(J) \leftarrow min[V(J_1), \ldots, V(J_n)]$

Minimax



Alternativas

No tenemos que generar todos los sucesores y guardar sus valores hasta que todos sean evaluados. Podemos usar un algoritmo *depth–first*:

Algoritmo *minimax* (*backtracking*)
si J es terminal, $V(J) \leftarrow ev(J)$, *sino*
for $k = 1, \ldots, n$ *do*
    generar J_k (el k–esimo sucesor de J)
    evalua $V(J_k)$
    si $k = 1$, $VA(J) \leftarrow V(J_1)$ *sino*
    para $K \geq 2$
        si J es nodo *max*, $VA(J) \leftarrow max[VA(J), V(J_k)]$
        si J es nodo *min*, $VA(J) \leftarrow min[VA(J), V(J_k)]$
regresa $V(J) \leftarrow VA(J)$

Alternativas

Tampoco es necesario que evaluemos todos los nodos terminales

Algortimo mejorado (ilustrado con *gana–pierde*)
si J es terminal, regresa, *gana/pierde*
empieza a etiquetar los sucesores de *J* (de izq. a der.)
si J es *max*, regresa *gana* en cuanto se encuentre un sucesor con *gana* o regresa *pierde* si todos los sucesores de *J* son *pierde*
si J es *min*, regresa *pierde* en cuanto se encuentre un sucesor con *pierde* o regresa *gana* si todos los sucesores de *J* son *gana*

Mejoras

- Las mejoras dependen del orden en que los sucesores son evaluados.
- Una de las razones de la eficiencia del último algoritmo es que se da cuenta que el estatus de algunos nodos no afecta la evaluación del nodo raíz.

Alpha-Beta

- A McCarthy (56) se le considera el responsable de ver la utilidad de *alpha–beta*.
- Parecido a *dynamic–programming* en el sentido de que elimina caminos que no van a producir mejores resultados.
- Es el más usado en juegos.

Alpha-Beta

Es como un *minimax* haciendo backtracking, pero . . .

> *si al actualizar el valor minimax de un nodo, éste cruza cierto límite, entonces no hace falta hacer más exploración abajo de ese nodo; su valor (VA) se puede transmitir a su padre como si todos sus hijos hubieran sido evaluados.*

Los límites o cortes son ajustados dinámicamente.

Alpha-Beta

- *Límite–alpha*: El límite para un nodo *J* MIN es un límite inferior llamado *alpha*, igual al *valor más alto de todos los ancestros* MAX *de J*. La exploración de *J* termina en cuanto el valor actual VA es igual o menor a *alpha*.
- *Límite–beta*: El límite para un nodo *J* MAX es un límite superior llamado *beta*, igual al *valor más pequeño de todos los ancestros* MIN *de J*. La exploración de *J* termina en cuanto el valor actual VA es igual o mayor a *beta*.

Alpha-Beta

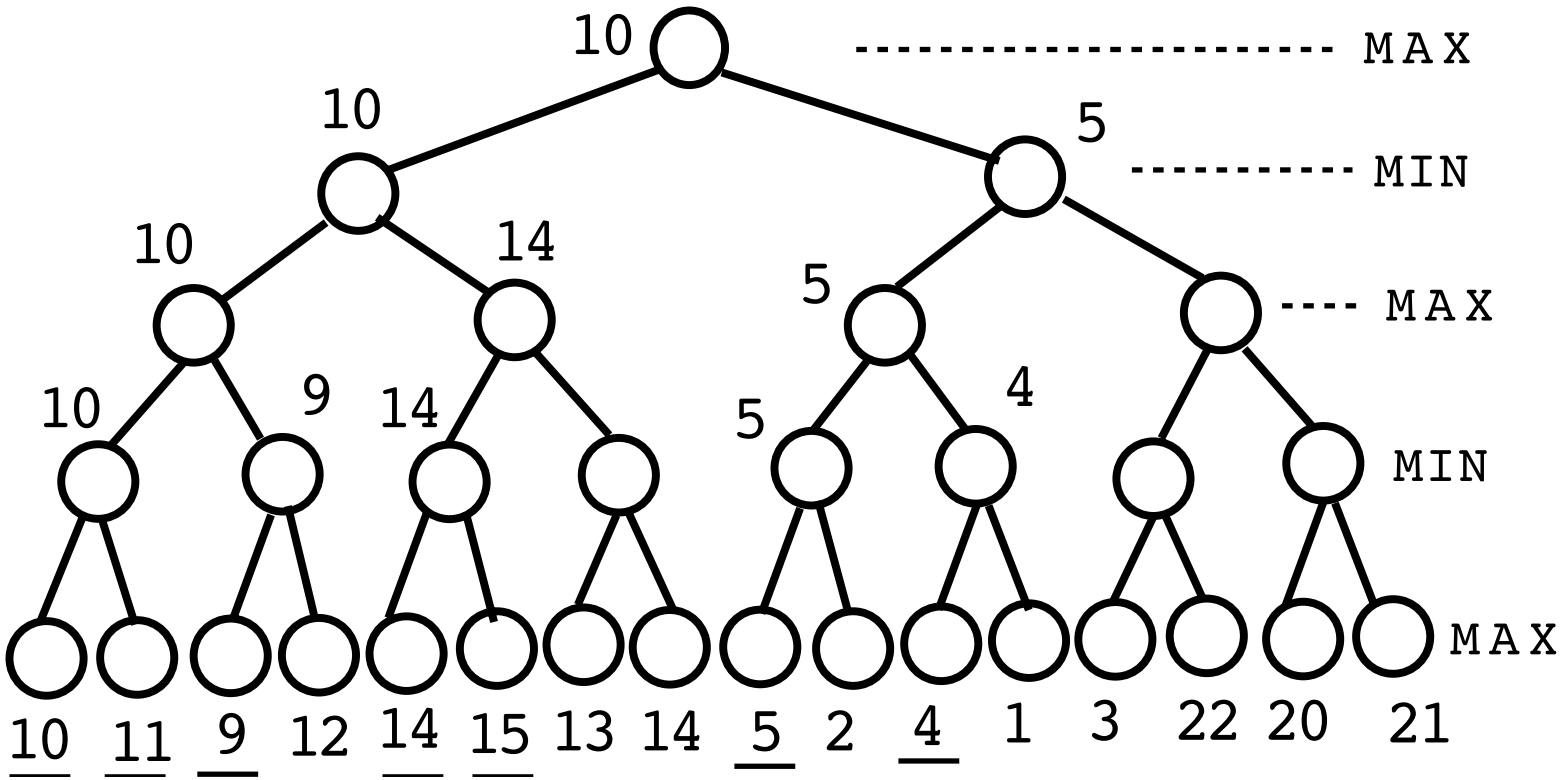
α representa el mejor valor que hemos encontrado hasta ahora para MAX y β el mejor valor (más bajo) para MIN.
Idea:

- Regresa el valor $V(J)$ si está entre α y β.
- Regresa α si $V(J) \leq \alpha$.
- Regresa β si $V(J) \geq \beta$.

Algoritmo Alpha-Beta

set $\alpha = -\infty$, *set* $\beta = \infty$
si J es nodo terminal, *entonces* $V(J) \leftarrow eval(J)$
sino sean $J_1, J_2, \ldots, J_n$ los sucesores de J
    set $k = 1$
    si J es *max*
        set $\alpha \leftarrow max[\alpha, V(J; \alpha, \beta)]$
        si $\alpha \geq \beta$ regresa β, *sino* continua
        si $k = n$ regresa α,
        sino set $k \leftarrow k + 1$ y continua
    si J es *min*
        set $\beta \leftarrow min[\beta, V(J; \alpha, \beta)]$
        si $\beta \leq \alpha$ regresa α, *sino* continua
        si $k = n$ regresa β,
        sino set $k \leftarrow k + 1$ y continua

Alpha-Beta



Juegos

- Se puede hacer una definición más concisa usando *neg–max*.
- La eficiencia de *alpha–beta* depende del orden de los valores de los nodos terminales.
- Si construimos un programa que pueda evaluar 1,000 posiciones por segundo, con 150 segundos por movida, podríamos ver 150,000 posiciones.
- Con un factor de arborescencia de 35 nuestro programa podría ver solo 3 o 4 tiradas (*ply*) adelante y jugaría ajedrez a nivel de novato.

Alpha-Beta

- Si pudieramos ordenar los nodos terminales tendríamos que examinar $O(b^{d/2})$, por lo que el factor de arborescencia efectivo es de $\sqrt{b}$ en lugar de b, en ajedrez sería 6 en lugar de 35, por lo que ahora podríamos buscar hasta profundidad 8.
- A veces se usan ordenamientos sencillos, como considerar primero las capturas, luego las amenazas, luego movimientos hacia adelante y finalmente movimientos hacia atrás.
- Otra estrategia es usar una búsqueda de profundidad iterativa y usar los últimos valores para decidir qué explorar en la siguiente iteración.

Alpha-Beta

- En promedio $\alpha - \beta$ permite explorar un 33% más que un simple *minimax*.
- $\alpha - \beta$ poda el árbol al darse cuenta que los nodos podados (independientemente de sus valores) no pueden influenciar en el valor del nodo raíz.

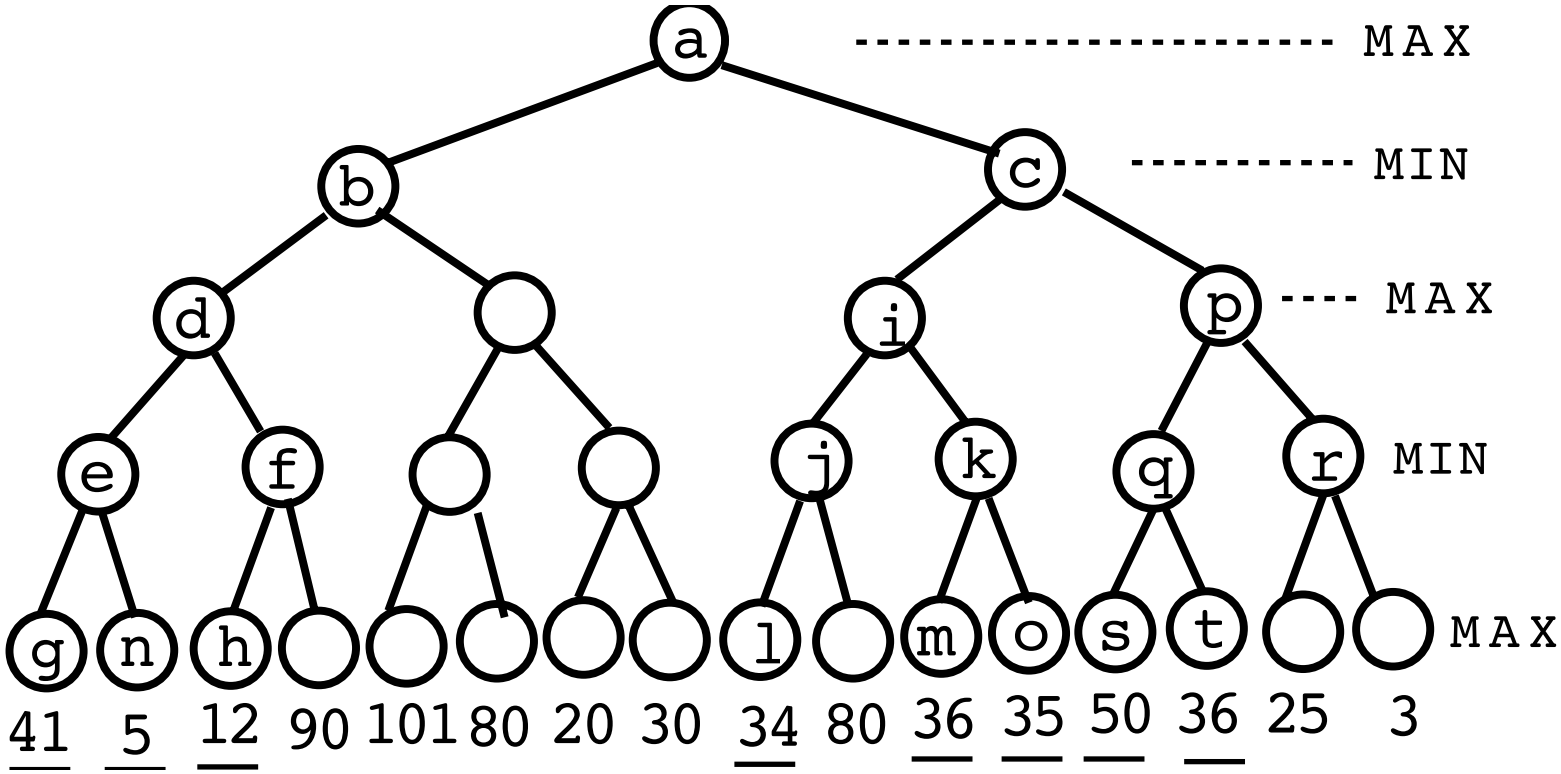
SSS* (Stockman 79)

Explora caminos tipo *best–first* (AO*).
Es superior a *alpha–beta* en el sentido que:

- No explora nodos que *alpha–beta* no explora.
- Puede no explorar nodos que *alpha–beta* si explora.

Idea: Tomar el árbol de búsqueda globalmente, se puede ver como una versión de *AO** y por lo mismo trata de encontrar la solución óptima.

SSS*



SSS*

- La figure muestra un ejemplo 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.
- Se han propuesto varias pequeñas variantes de SSS* como InterSSS*, RecSSS* y Mem*, las cuales hacen un uso más eficiente de memoria.
- Existen muchos otros algoritmos de juegos, por ejemplo SCOUT (Pearl 80)

Estrategias de Juego

- *Progressive/iterative deepening*: explorar por niveles.
- Si: b = *branching factor* y d = profundidad: En general el número de nodos evaluados al final es: b^d.
- El número de nodos en el resto del árbol es:

$$\sum_{i=0}^{d-1} b^i = \frac{b^d - 1}{b - 1}$$

Estrategias de Juego

La razón entre ellos es:

$$b^d \frac{b-1}{b^d-1} \approx b-1$$

i.e., con profundidad de 16 el costo total es $\frac{1}{15}$ del costo normal lo cual es bastante bajo.

Estrategias de Juegos

- *Heuristic Pruning*: Ordenar los nodos con respecto a su función de evaluación y variar la profundidad de exploración (mayor/menor). Sin embargo con sólo esta estrategia no habría sacrificios de damas en ajedrez.
- *Heuristic Continuation*: Continuar caminos que por heurísticas se consideran importantes (e.g., el rey peligra, se va a perder una pieza, un peón puede coronar, etc.), para tratar de evitar el *efecto horizonte* (no siempre se puede).

Estrategias de Juegos

- *Quiescence Search*: En principio, no se debería de aplicar la función de evaluación en posiciones en donde no se esperan cambios fuertes en el valor de la función de evaluación.
- A veces se restringe sólo a algunos tipos particulares de movidas (e.g., capturas).
- Todas las metodologías suponen que la decisión de la jugada mejora con la profundidad de la búsqueda.

Estrategias de Juegos

- Se ha mostrado que funciona bien en juegos y ésto se cuestiona poco. De hecho se cree que el nivel extra de exploración en ajedrez mejora en 200 el puntaje de la máquina.
- Sin embargo, se ha mostrado que el nivel discriminatorio decrece apreciablemente en cada nivel.
- Si la función de evaluación se mantiene igual en todos los niveles de juego, entre más buscamos a profundidad, peor es nuestra evaluación.

Estrategias de Juegos

Porqué no ocurre en los juegos?

> *Los juegos comunes no son uniformes, están llenos de "trampas" que son detectadas al buscar a profundidad.*

Sin embargo, no se debe de perder de vista el deterioro del algoritmo.

Estategias de Juegos

- Algo muy usado en juegos (además de guardar aperturas y finales de juego) son las *transposition tables*. Esto es, guardar información de nodos recorridos para evitar recorrerlos otra vez.
- Existen muchas otras estrategias, e.g., B* (Berlinger 79)
- Líneas de investigación: Usar *meta*–razonamiento y combinar planeación con búsqueda.

MCTS (*Monte Carlo Tree Search*)

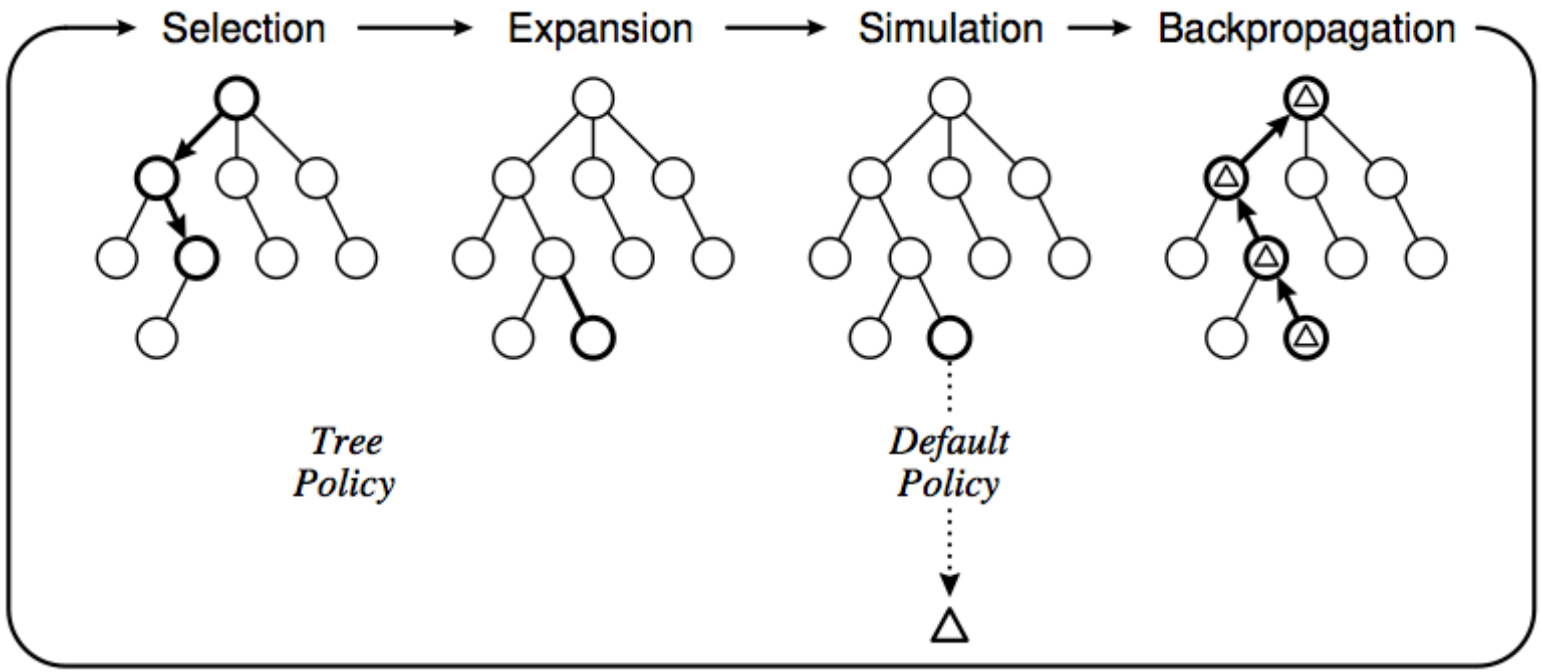
- Es un método para encontrar decisiones óptimas tomando muestras aleatorias y construyendo un árbol de búsqueda de acuerdo a los resultados
- Desató recientemente un gran interes dados los resultados que se empezaron a obtener en el juego de Go
- Idea: Analizar los movimientos más promisorios expandiendo el árbol de búsqueda haciendo muestreo aleatorio

MCTS

Cada ronda de MCTS consiste de 4 pasos:

1. Selección: Empezando de la raíz se seleccionan hijos sucesores hasta llegar a una hoja (cómo seleccionar ver abajo)
2. Expansión: A menos que la hoja termine el juego, crea sus hijos y seleciona uno de ellos
3. Simulación: Empieza un juego aleatorio de ese nodo (*random playout*) - en un *playout* el juego se lleva hasta el final seleccionado acciones aleatorias
4. Retropropagación: Usando la información del *playout* se actualiza la información en los nodos del camino del nodo raíz al nodo desde empezó la simulación

MCTS



UCT (*Upper Confidence Bound applied to Trees*)

- Un aspecto fundamental de los MCTS es cómo balancear exploración y explotación.
- Esto es, explotación de variantes profundas después de movimientos con promedios altos y la exploración de movimientos con pocas simulaciones
- Un muestreo es selectivo puede hacer grandes ahorros de cómputo
- La idea de UCT es aplicar ideas de problemas *multi-armed bandit* (UCB1) para guiar cómo explorar/explotar ese árbol

UCT

La fórmula básica de UCT es seleccionar en cada nodo del árbol, la movida para la cual la expresión siguiente tenga el valor máximo:

$$\text{argmax}_{v' \in hijos(v)} \frac{Q(v')}{N(v')} + c\sqrt{\frac{\ln N(v)}{N(v')}}$$

donde:

- $Q(v')$: Recompensas totales (el número de veces que se gana después del i-ésimo movimiento)
- $N(v')$: Número de veces que se ha visitado (simulaciones) el i-ésimo movimiento
- $N(v)$: Número total de simulaciones que se han hecho en el nodo (igual a la suma de todas las v's)
- c: Parámetro de exploración, en teoría igual a $\sqrt{2}$, pero en la práctica seleccionado empíricamente

UCT

Ventajas:

- No requiere la definición de una función de evaluación (no siempre es fácil definir una)
- El árbol de juego crece asimétricamente, ya que se concentra en las partes más promisorias, con lo que le va bien en juegos con factores de arborecencia grandes
- MCTS se puede parar en cualquier momento (*any-time*)

UCT

- Los *playouts* pueden ser “light” (usando movidas aleatorias) o “heavy” (usando heurísticas para seleccionar las movidas)
- Para las estadísticas se pueden aprovechar pedazos de juegos que se repitan.
- Por ejemplo en GO algunas jugadas/posiciones se repiten varias veces en el juego y están relativamente aisladas de las demás, por lo que se pueden aprovechar todas las estadísticas de éstas.

UCT

Existen varias versiones paralelas:

- Paralelización de hojas: Ejecutar varios *playouts* en paralelo
- Paralelización de raíz: Construir varios árboles en paralelo y realizar el movimiento basado en las ramas de nodo raíz de todos los árboles
- Paralelización del árbol: Construir en paralelo el árbol de juego cuidando de posibles conflictos

AlphaGo

- Existen varias razones por las cuales Go resulta ser mucho más difícil que ajedrez

	Ajedrez	Go
Tamaño del tablero	8×8	19×19
Movidas promedio por juego	80	300
Factor promedio de arborecencia	35	235
Espacio de búsqueda	10^{50}	10^{170}

- Es difícil de definir una función de evaluación adecuada para Go

AlphaGo

AlphaGo hace una combinación de DNN, MCTS, SL y RL

- Aprende primero una red para predecir los movimientos de jugadores humanos usando aprendizaje supervisado usando 30 millones de posiciones y una red de 13 capas.
- Después usa RL para mejorar la red aprendida (esta mejora gana 80% de los juegos contra la red anterior).
- Se construye otra red para evaluar posiciones. Esto es, regresar un valor para cada posición (en lugar de una distribución de probabilidades). Se usan 30 millones de posiciones nuevas.
- AlphaGo combina la red de la política (la mejorada) con la red para evaluar posiciones en un MCTS.

Juegos con eventos externos

- Juegos, como *backgammon*, incluyen factores externos no predecibles (e.g., el resultado de tirar un dado).
- Para este tipo de juego, se incluyen nodos intermedios (*chance nodes*) que contemplan los posibles resultados de los eventos externos con sus probabilidades.
- Para propagar valores, de los nodos terminales se toman sus valores y al llegar a los nodos “aleatorios” se calcula su valor esperado sumando todas las posibilidades multiplicadas por su probabilidad para obtener un valor.

Hombres vs. Máquinas

Temores:

- Juegos aburridos
- Jugadas sin sentido para el hombre

Hombres vs. Máquinas

Estado del arte:

- Ajedrez: Kasparov es derrotado en condiciones de torneo por Deep Blue (1997).
  Deep Blue (97): 32 procesadores RS/6000 (Power Two Super Chip (P2SC)), cada uno con 8 procesadores VLSI de ajedrez, para un total de 256 procesadores. Analiza alrededor de 100–200 mil millones de posiciones cada 3 minutos y alrededor de 200,000,000 posiciones por segundo. Kasparov analiza aproximadamente 3.
- Damas Chinas: Chinook es declarado el campeón mundial cuando Tinsley se retira por razones de salud (1994).

Hombres vs. Máquinas

- Backgammon: TD-gammon es considerado como uno de los 3 mejores jugadores del mundo (1995). Este sistema usa aprendizaje por refuerzo
- Othello: Logistello considerado muy superior a los humanos (1994).
- Atari: Un programa basado en DL + RL aprende a jugar todos los juegos de Atari, algunos a nivel expertos (2016)
- Go: Alpha-Go le gana al campeón mundial (2017)