Capítulo 2

Procedimientos de Búsqueda Clásicos

2.1 Introducción

Primero, algunos conceptos: nodos, arcos, nodo inicial, sucesor o hijo, padre, grado de arborecencia (asumimos finito), árbol, nodo raíz, hoja o nodo terminal, árbol uniforme, pesos/costos/premios asociados, camino, decendiente, ancestro.

- Nodos expandidos (CLOSED)
- Nodos explorados pero no expandidos
- Nodos generados pero no explorados (OPEN)
- Nodos no generados

Es común asociar una estructura de datos para cada nodo, por ejemplo:

- El estado en el espacio de estados al que corresponde el nodo
- El nodo que generó ese nodo (su padre)

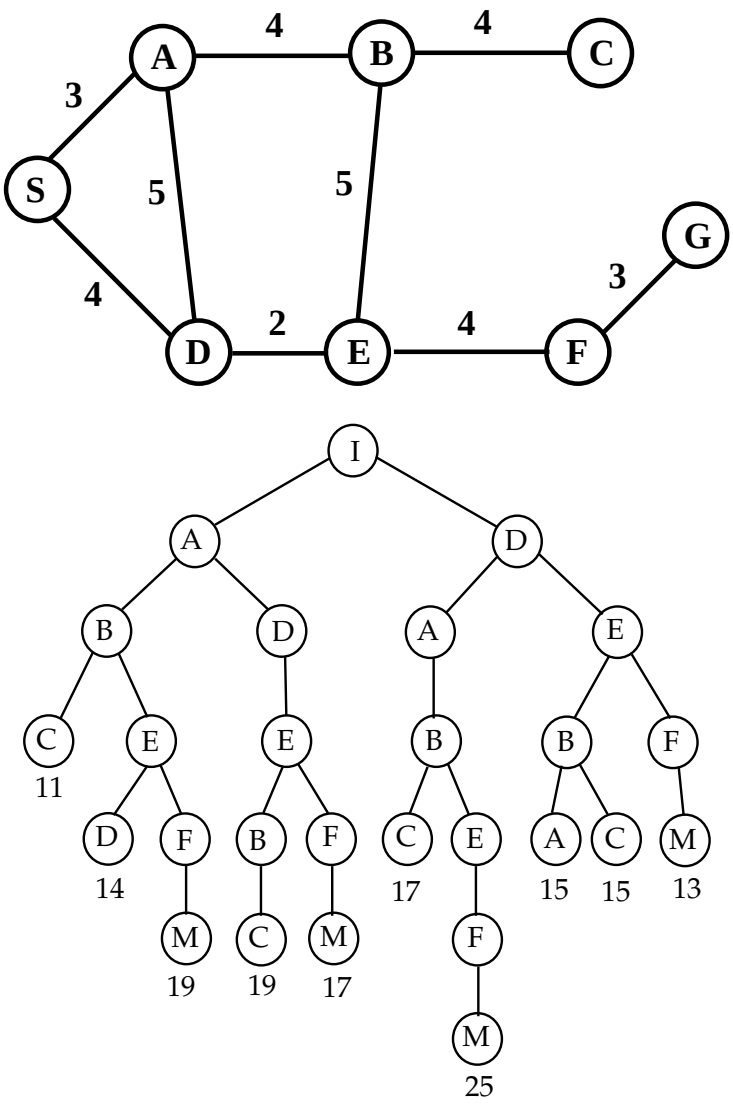

Figura 2.1: Un grafo y su corresondiente árbol de búsqueda de ejemplo

- El operador que se uso para generarlo
- El número de nodos en el camino de la raíz al nodo (profundidad del nodo)
- El costo del camino

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
3. *Dominante*: un algoritmo A_1 se dice que domina a A_2 si todo nodo expandido por A_1 también es expandido por A_2 ("más eficiente que"). Se dice extrictamente dominante si A_1 domina a A_2 y A_2 no domina a A_1.
4. *Óptimo sobre una clase*: un algoritmo se dice óptimo sobre una clase de algoritmos si domina a todos los miembros de la clase.

2.2 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.

2.2.1 Breadth–first search (búsqueda a lo ancho)

Explora progresivamente en capas del grafo de misma profundidad.

La forma de implementarlo es poner los sucesores de cada nodo al final de una cola o agenda, por lo que OPEN (lista de nodos por explorar) se implementa como un *stack*.

Tabla 2.1: Algoritmo Genérico 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

Tabla 2.2: Algoritmo Breadth-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 y
        añade sus sucesores al *final* de la agenda

Tabla 2.3: Requerimientos de tiempo y memoria para breadth-first. Factor de arborecencia = 10; 1,000 nodos/sec; 100 bytes/nodo.

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

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. Básicamente tiene que guardar la parte completa de la red que está explorando.

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: $1 + b + b^2 + b^3 + \ldots + b^d$

Los requerimientos de memoria es un problema grande para breadth–first.

También el tiempo es un factor importante. Básicamente problemas de búsqueda de complejidad exponencial no se pueden resolver, salvo para sus instancias más pequeñas.

2.2.1.1 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

Tabla 2.4: Algoritmo Depth-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 y
        añade sus sucesores al *frente* de la agenda

nunca decrece al aumentar los caminos.

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

2.2.2 Depth–first o Búsqueda en Profundidad (LIFO)

Depth–first siempre expande uno de los nodos a su nivel más profundo y sólo cuando llega a un camino sin salida se regresa a niveles menos profundos.

En depth–first cada nodo que es explorado genera todos sus sucesores antes de que otro nodo sea explorado. Después de cada expansión el nuevo hijo es de nuevo seleccionado para expansión.

Si no se puede continuar, se regresa al punto más cercano de decisión con alternativas no exploradas.

Se puede implementar añadiendo los sucesores de cada nodo al frente de la agenda.

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.

Su complejidad en tiempo (cuanto se tarda) es $O(b^m)$ en el peor de los casos.

Para problemas con muchas soluciones depth-first puede ser más rápido que breadth–first porque existe una buena posibilidad de encontrar una solución después de explorar una pequeña parte del espacio de búsqueda

Depth–first no es ni completo ni óptimo y debe de evitarse en árboles de búsqueda de profunidad muy grande o infinita.

2.2.2.1 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).

2.2.2.2 Búsqueda con Profundidad Limitada (depth–limited)

Para evitar caer en caminos de profunidad muy grande, a la mayoría de los algortimos que usan depth–first se les impone un límite de profundidad máxima.

Si se sabe la profundidad de alguna meta, el algoritmo puede ser completo, pero sigue sin ser óptimo. Si la profunidad es l, se tarda $O(b^l)$ y ocupa $O(bl)$.

Cuando se tienen grafos altamente conectados hay que tener cuidado con el concepto de profundidad (una más que el padre menos profundo). Por lo que en este caso, se tienen que analizar todos los padres, lo cual implica requerimientos adicionales de memoria.

En la práctica se puede tomar solo la profundidad del padre que lo generó haciendo una estrategia LIFO más "pura".

2.2.2.3 Búqueda 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.

La cantidad de nodos expandidos es similar a breadth–first, solo que algunos nodos se expanden varias veces.

Aunque parece que se repite mucho trabajo de más, en la práctica es relativamente poco.

Por ejemplo, en lugar de que sea $1 + b + b^2 + ... + b^d$, el primero se expande $d + 1$ veces, el segundo d veces, el tercero $d - 1$, etc.

Por ejemplo, con un *branching factor* de 10 y con profundidad 5, los nodos expandidos serían $1 + 10 + 100 + \ldots + 100,000 = 111,111$, 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.

Sin embargo, inclusive para un árbol binario se toma como el doble de tiempo que breadth–first pero con mucho menos memoria.

En general, si el espacio de búsqueda es grande y no se sabe la profundidad del árbol es el método a usar.

2.2.3 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.

Tabla 2.5: 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

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 (i.e., se tienen una descripción del conjunto posible de estados), lo cual lo vuelve mucho más difícil. Por ejemplo, cuáles son los estados predecesores de una condición de jaque mate en ajedrez?
- 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).

Para revisar si los nodos se encuentran, por lo menos se tienen que guardar todos los nodos de una mitad (como en breadth–first).

2.2.4 Evitando Estados Repetidos

Hasta ahora hemos ignorado la posibilidad de expader nodos que ya visitamos en algún camino. En muchos problemas, no podemos evitar el tener estados repetidos, en particular, todos los problemas con operadores reversibles.

Los árboles de búsqueda en este caso pueden ser inifinitos, pero si eliminamos los repetidos los podemos volver finitos.

Opciones:

- No regresar al estado anterior.
- No crear caminos con ciclos
- No generar algun estado que fué generado antes. Esto requiere guardar en memoria todos los estados generados.

Los algoritmos de búsqueda normalmente usan una tabla hash, y el qué tanto guardamos depende muchas veces del problema. Entre más ciclos tenga es más probable que el evitar generar estados ya generados sirva más.

2.2.5 Búsqueda en grafos AND/OR

Tanto breadth–first como depth–first pueden ser adaptados para buscar en grafos AND/OR. Las diferencias están sobretodo en determinar las condiciones de terminación.

En lugar de involucrar las propiedades de un solo nodo, pueden involucrar un conjunto de nodos.

Se puede usar el algoritmo de propagación de etiquetas:

- Si es nodo terminal, etiquétalo
- Si en nodo OR no terminal, etiquétalo si existe al menos una etiqueta
- Si en nodo AND no terminal, etiquétalo si todos estan etiquetados

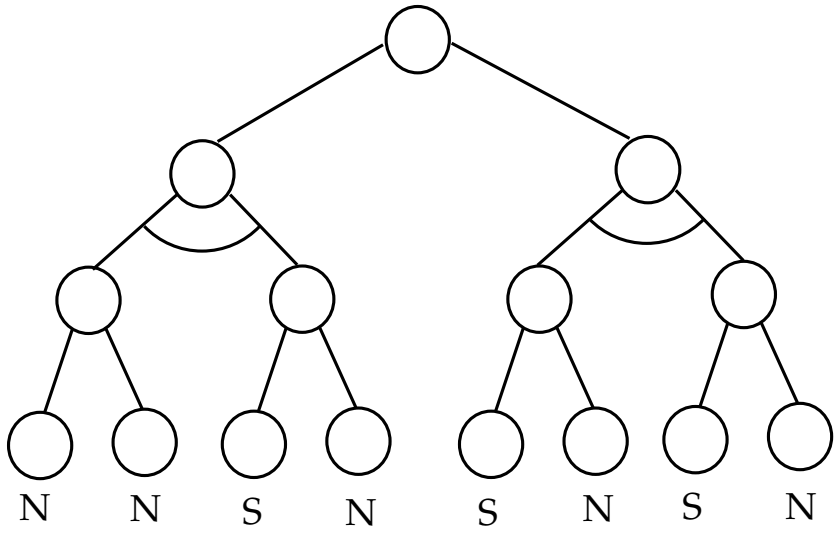

Figura 2.2: Arbol And – Or

Breadth–first transmite el impacto a todo el grafo, backtracking transmite a su camino de travesia (*traversal path*)

A veces, algunos subproblemas se encuentran repetidos en el grafo. Para ésto se tendría que guardar sus soluciones para aprovecharlas o volver a generar las soluciones.

2.3 Búsqueda con Información

Normalmente la información se incluye al momento de decidir qué nodo expander por medio de una función de evaluación.

2.3.1 Hill–Climbing

Hill–climbing es una estrategia basada en optimización local.

Tabla 2.6: 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
```

Tabla 2.7: Algoritmo Hill-climbing

```
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
```

Sigue la dirección de ascenso/descenso más empinada a partir de su posición y requiere muy poco costo computacional.

Se llama también una estrategia irrevocable porque no permite regresarnos a otra alternativa.

Es útil si se tiene una función heurística muy buena o cuando los operadores de transición entre estados tienen cierta independencia (conmutativa), que implica que la operación de un operador no altera la futura aplicación de otro.

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*

Tabla 2.8: Algoritmo 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

hill–climbing), salvando el mejor (v.g., GSAT).

Dado su bajo costo computacional es la estrategia de búsqueda más utilizada en optimización y aprendizaje.

2.3.2 Best–First

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

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

Para ésto, se toman en cuenta todos los nodos que se han visto hasta el momento.

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.

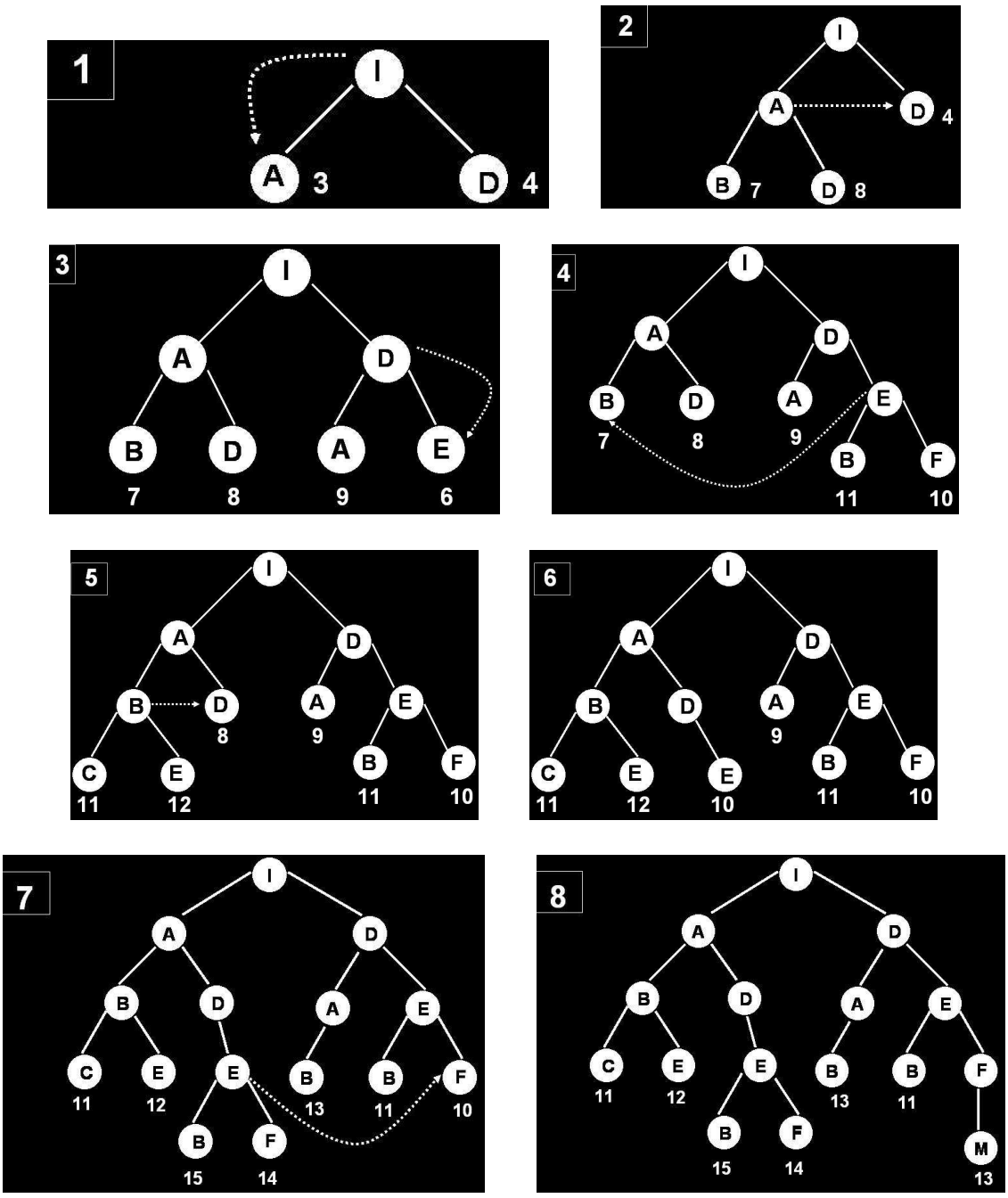

Figura 2.3: Búsqueda Best first

Una estrategia "greedy" es susceptible de errores. El tiempo en el peor de los casos es $O(b^m)$ donde m es la profunidad máxima del espacio.

Debido a que guardan todos los nodos en memoria, su complejidad en espacio es igual a la del tiempo.

2.3.3 GBF (o BF para grafos AND/OR)

Si queremos aplicar best–first en grafos, tenemos que tener cuidado en cuanto a la definición del "mejor" y de "candidato", ya que tenemos conjuntos de nodos a considerar y cada nodo puede estar en más de un conjunto.

Para garantizar que el algoritmo GBF es óptimo los costos asociados deben de ser optimistas (subestimaciones) y se debe de modificar la condición de terminación, para que en lugar de quedarse con la primera solución, hay que verificar que las demás posibilidades sean realmente peores.

2.3.4 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)

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)

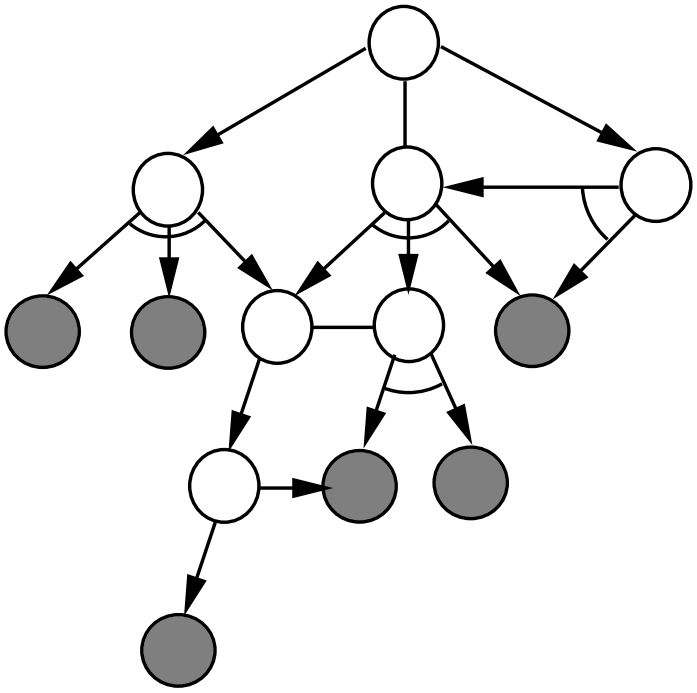

Figura 2.4: Ejemplo de grafo And – Or

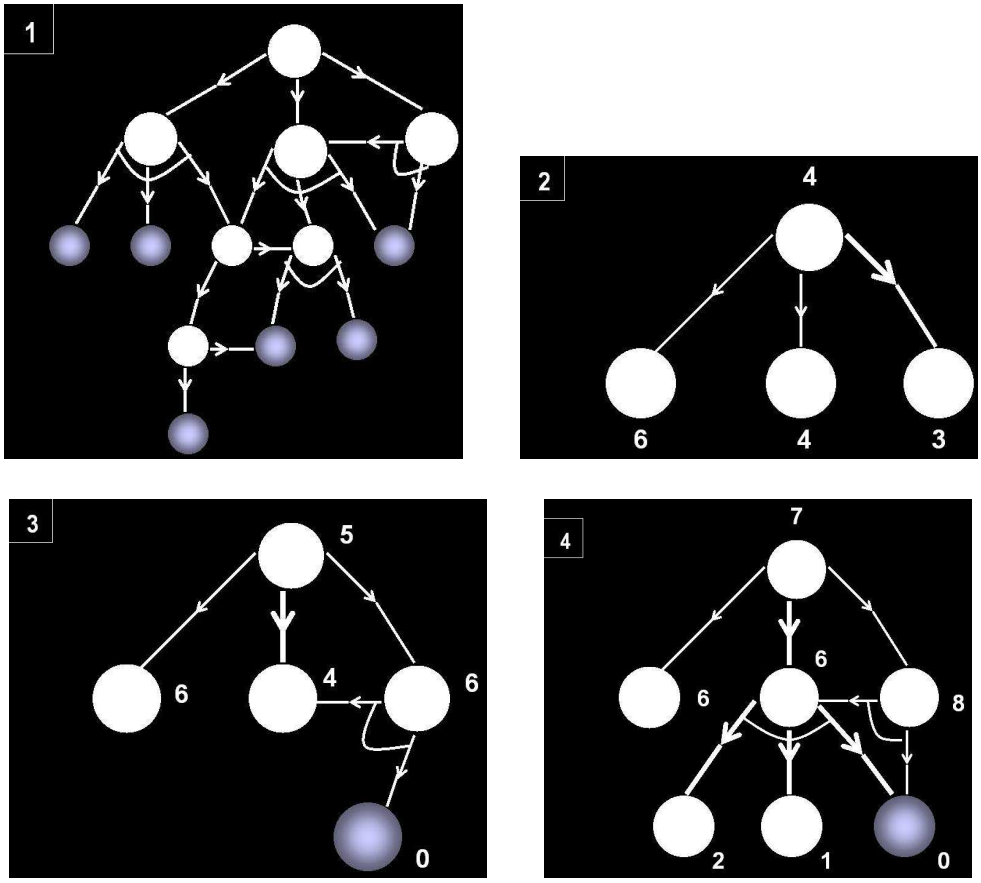

Figura 2.5: Búsqueda GBF

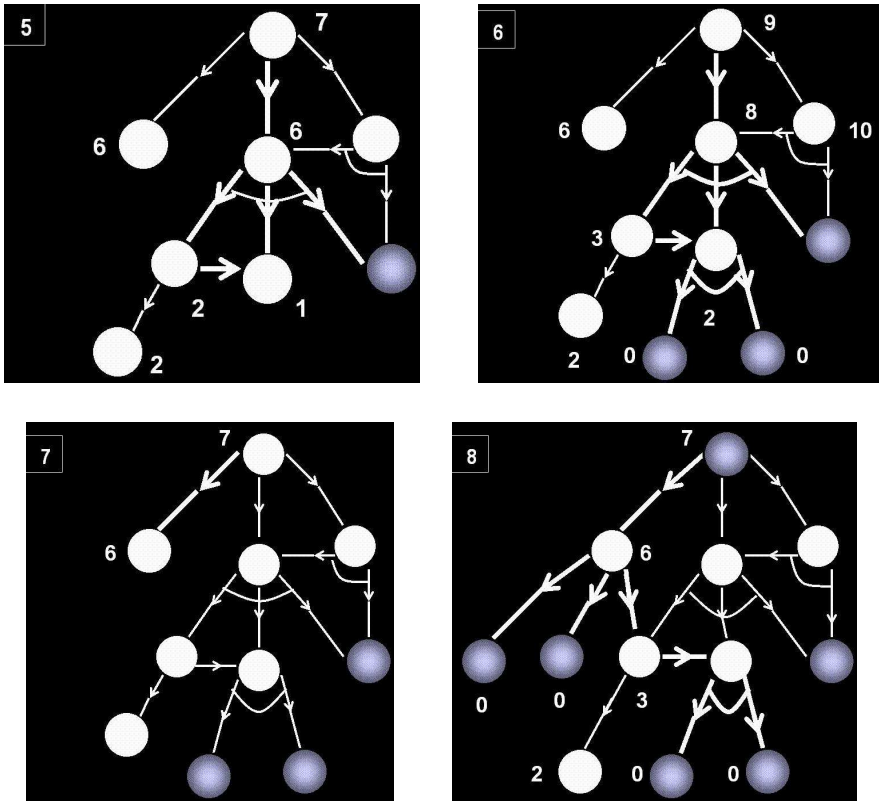

Figura 2.6: Búsqueda GBF (cont.)

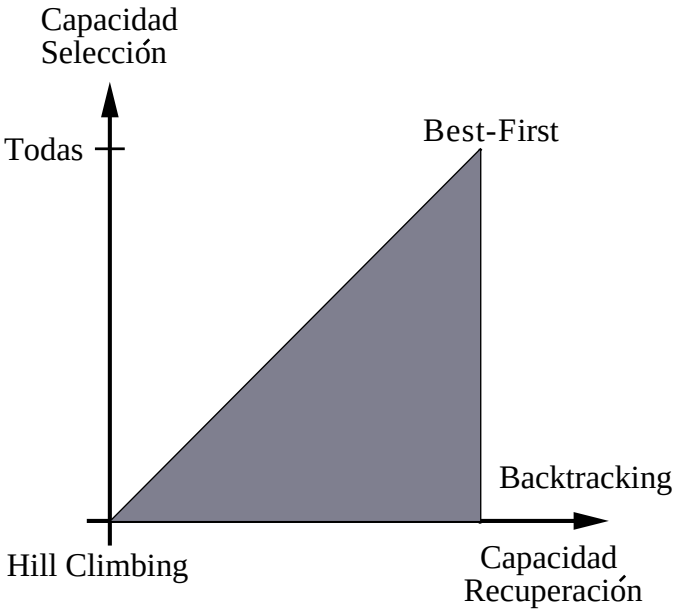

Figura 2.7: Algoritmos Híbridos

Tabla 2.9: Algoritmo 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)

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.

2.3.5 Beam–Search

Otra alternativa, que está entre hill–climbing y best–first es beam–search. En esta estrategia, se mantienen las k mejores alternativas.

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

2.3.6 Mejor Solución

Cuando importa el costo de encontrar una solución.

Si $g(P)$ es el costo de camino o solución parcial, la solución óptima es aquella con $g(P)$ mínima.

Una forma segura: búsqueda exhaustiva y seleccionar el de menor costo

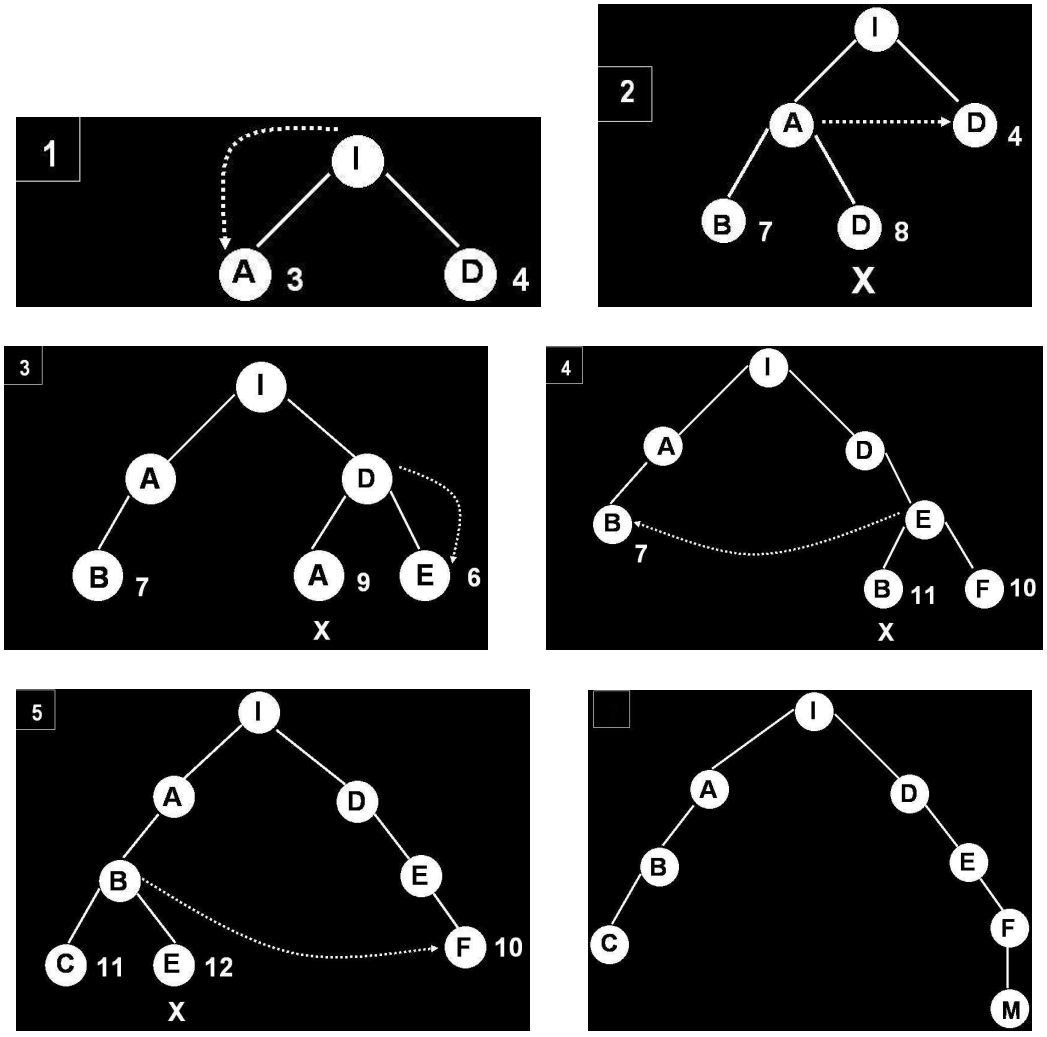

Figura 2.8: Búsqueda Beam search con $k = 2$.

Tabla 2.10: Algoritmo Branch and Bound

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

(Brittish Museum).

Best–first no es admisible, pero con una pequeña variante ya lo es.

2.3.7 Branch and Bound

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

2.3.8 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*

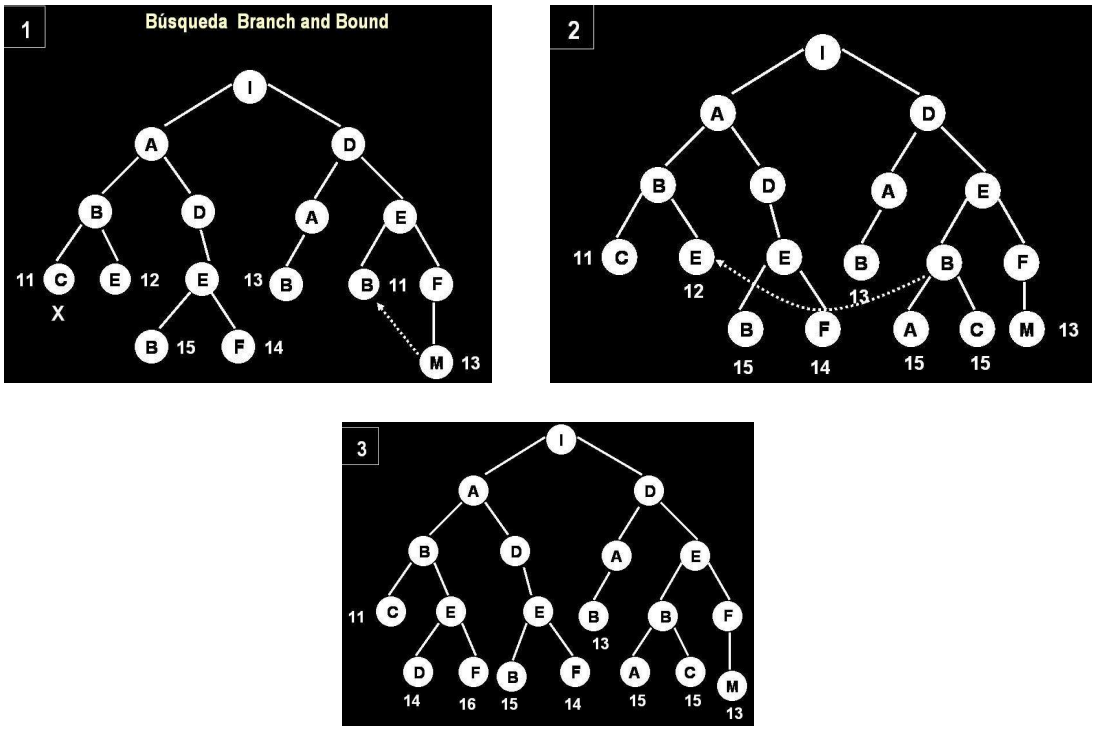

Figura 2.9: Búsqueda Branch and Bound

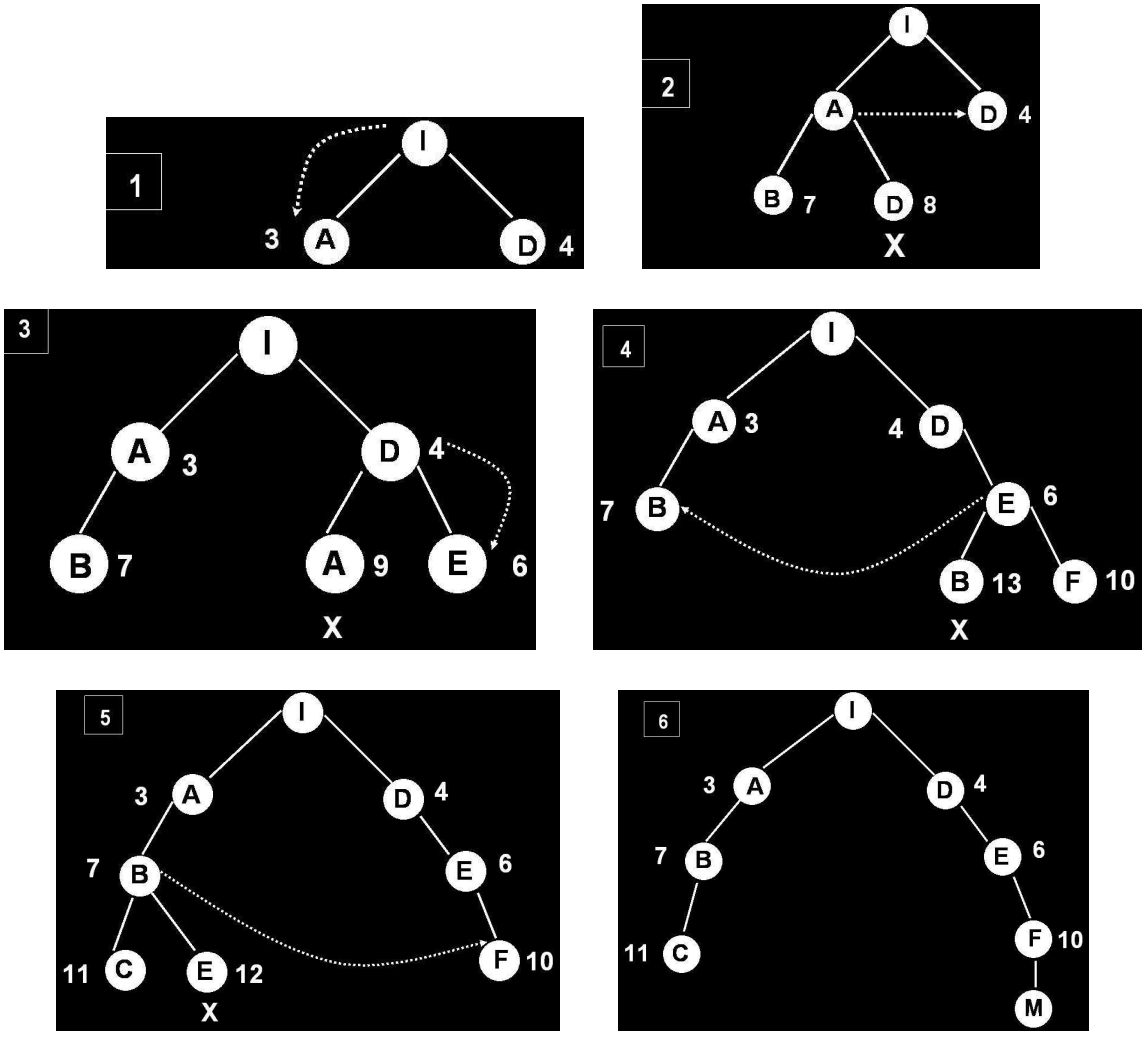

Figura 2.10: Búsqueda con programación dinámica

Tabla 2.11: 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
```

2.3.9 A* (Minimizando el costo total del camino)

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. Una función es admisible si $\forall n\ h(n) \leq h^*(n)$.

Propiedades de A*:

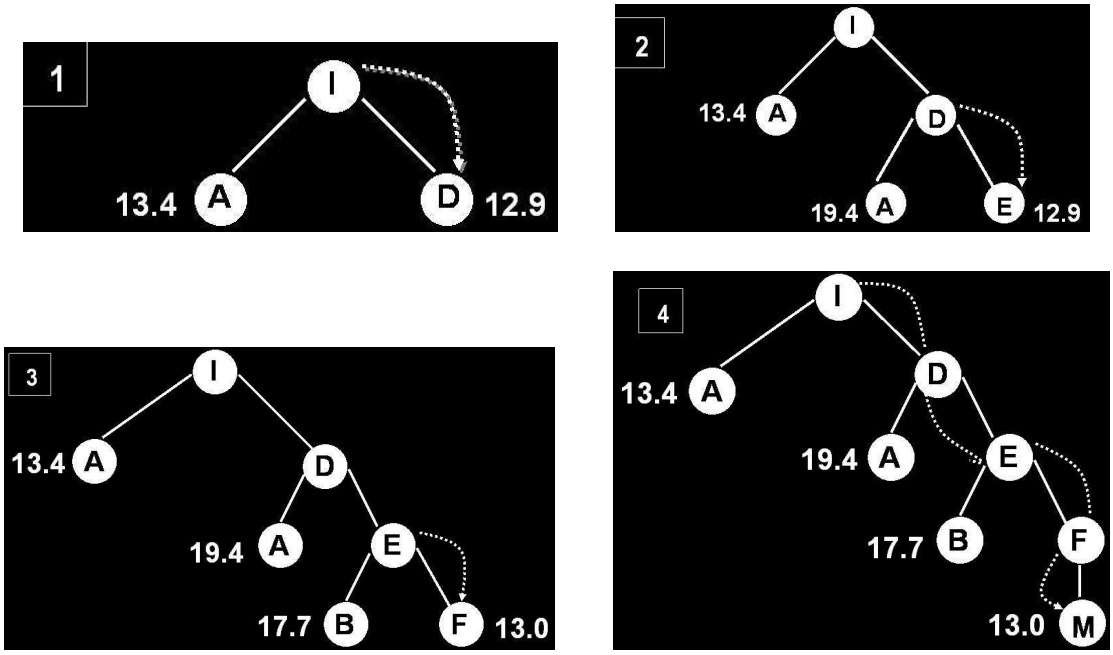

Figura 2.11: Búsqueda A*

- A* siempre termina en grafos finitos
- A* es completa en grafos finitos
- Si $h(n)$ es un estimador optmista de $h^*(n)$, A* regresa la solución óptima
- A* es admisible

Una heurística $h(n)$ se dice *consistente* si:

$$\forall(n, n')\ h(n) \leq k(n, n') + h(n')$$

donde $k(n, n')$ es el costo más barato del camino entre 2 nodos.

Cualquier heurística consistente es también admisible.

Una heurística $h(n)$ se dice *monotona* si:

$$h(n) \leq c(n, n') + h(n')\ \forall(n, n') \mid n' \in SCS(n)$$

donde $c(n, n')$ es el costo entre dos nodos y $SCS(n)$ son los sucesores de n. Esto es el costo nunca decrece.

Se puede probar que una heurística es monotónica si obedece la desigualdad del triangulo, osea que la suma de sus lados no pueden ser menores a el otro lado.

De hecho, las dos propiedades son equivalentes.

Si tenemos heurísticas no monotónicas, podemos usar el costo del padre para garantizar que el costo nunca decrece: $f(n') = max(f(n), g(n')+h(n'))$. Esto se llama *pathmax*.

Si f nunca decrece al recorrer los caminos, se pueden "dibujar" contornos ("curvas de nivel").

Si $h = 0$ (costo uniforme) las bandas con "circulares" alrededor del estado inicial.

Con heurísticas más precisas, las bandas tienen a alargarse hacia las metas.

A* va expandiendo todos los nodos en estos contornos hasta llegar al contorno de la meta, en donde posiblemente expanda algunos antes de llegar a la solución.

Básicamente si no se expanden todos los nodos de un contorno se puede perder la solución óptima.

A* es completo, óptimo y eficientemente óptimo (ningún otro algoritmo óptimo garantiza expander menor nodos que A*).

Sin embargo, el número de nodos normalmente crece exponencialmente con la longitud de la solución, a menos que el error en la solución no cresca más rápido que el logaritmo del costo actual del camino.

$$| h(n) - h^*(n) | \leq O(log(h^*(n)))$$

Generalmente A* se queda sin espacio (antes de consumir el tiempo).

Si se conoce un estimado del costo óptimo, se pueden podar ciertos nodos. Por ejemplo, si un depth–first encuentra una meta, ésta nos puede servir para eliminar nodos con costos mayores.

Usualmente el factor de arborecencia efectivo de una heurística se mantiene más o menos constante sobre una gran variadad de instancias del problema, por lo que medidas experimentales en pequeñas conjuntos pueden dar una guía adecuada de lo útil de la heurística.

Idealmente la heurísitca se debe de acercar a 1.

Una heurística (h_1) se dice que domina a otra (h_2) si h_1 expande en promedio menos nodos que h_2.

A* expande todos los nodos con $f(n) < f^* \equiv$ expande todos los nodos que cumplan $h(n) < f^* - g(n)$. Como h_1 es por lo menos tan grande que h_2 para todos los nodos, todo nodo expandido por h_1 también es expandido por h_2 (quien posiblemente expande otros).

Por lo que siempre es mejor usar la heurística que de valores más grandes, siempre y cuando no sobre–estime.

Tabla 2.12: Comparación Interactive Deepening (IDS) con A* con dos heurísticas (h_1 y h_2). 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

2.3.10 Extensiones

En la práctica se ha visto que a veces A* se pasa mucho tiempo explorando caminos cuyos costos no varían demasiado.

En estos casos, la propiedad de admisibilidad empieza a ser más un problema que una virtud, ya que evita encontrar rápidamente soluciones que a pesar de no ser óptimas se pueden acercar bastante al óptimo.

Posibilidades:

La fórmula: $f = g + h$ viene de dos principios: (i) lo más corto es lo más rápido (g) y (ii) para asegurarnos que es la mejor opción hay que agregar subestimaciones (h).

El efecto de g es para añadir la componente de breadth–first a la búsqueda, mientras h es ideal cuando se tienen muchas soluciones y se puede confiar en la estimación.

Se han propuestos funciones pesadas para compensar cada parte:

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

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

Si $0 \leq w \leq 1/2$ es fácil garantizar la admisibilidad, pero se puede perder para $w > 1/2$, dependiendo de que tal alejada está h de h^*.

Una estrategia ϵ-admisible se obtiene con un peso dinámico: En lugar de mantener a w constante durante la búsqueda, mejor usar un peso dinámico que se ajusta con la profundidad. Una posibilidad es:

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

donde $d(n)$ es la profundidad del nodo n y N la profundidad anticipada del nodo meta. A profunidades bajas, h tiene un soporte de $1 + \epsilon$ lo que favorece

escursiones tipo depth-first, mientras que a grandes profundidades asume un peso igual.

También se han desarrollado medidas para estimar el riesgo asociado con dejar algun nodo sin explorar.

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

2.3.10.1 IDA*

Generalmente lo primero que se agota en A* es la memoria. Dos extensiones a A* tratan de reducir los requerimientos de memoria.

IDA*, es una extensión natural de *iteractive deepening*. Se usa un límite de costo (más que de profundidad) iterativo.

IDA* es completo y óptimo como A*, pero por ser depth–first, sólo requiere memoria proporcional al camino más largo que explora (aprox. bd, porqué depende del número de nodos por costo).

La complejidad de tiempo depende del número de valores que puede tomar la heurística.

Sin embargo, en problemas en donde la heurística cambia para cada estado (e.g., el TSP), lo que quiere decir es que cada contorno incluye sólo un estado. Si A* expande N nodos, IDA* tendría que hacer N iteraciones ($1+2+\ldots+N$), osea $O(N^2)$

Por lo que si N es muy grande para ser guardada en memoria, seguramente N^2 es demasiado tiempo que se necesita esperar.

2.3.10.2 A$^*_\epsilon$

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 de simplicidad de solución, permitiendo

errores del orden del rango establecido. Si el rango es igual a cero se regresa al algoritmo original.

Básicamente 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.

A^*_ϵ se comporta como A* solo que A^*_ϵ selecciona un nodo de FOCAL usando una heurística nueva que estima el esfuerzo computacional de los nodos en FOCAL para llegar a la solución.

Esto puede provocar que se regresen soluciones que no son óptimas con un error a lo más de ϵ (se llaman también ϵ–admissible)

Algo muy parecido es establecer un rango en donde se permite tener sobre–estimaciones.

2.3.10.3 MA*

Espacios de estados con una gran variedad de costos son difíciles porque el algoritmo extiende su frontera y cambia frecuentemente de oponión acerca de qué camino es el más promisorio.

Para A* no es tanto problema porque guarda todos, pero para IDA* es especialmente difícil porque no guarda información acerca de los caminos.

En problemas donde se tienen grafos altamente conectados, es muy probable que se repitan estados generados anteriormente.

Una posibilidad es guardar tantos nodos como sea posible (los mejores). Otra posibilidad es que cuando se elimine un nodo (que ya no cabe en memoria), se guarde la mayor cantidad de información acerca de su costo en su antecesor.

MA* hace mas o menos ésto. Al llenar su memoria elimina el nodo de mayor costo de su agenda y propaga su costo a su nodo padre. A pesar de eliminar la información que dice como ir a un cierto nodo, los valores guardados le sirven al algortimo para saber que tanto vale la pena explorar el nodo padre.

Durante el transcuros del tiempo, los nodos guardados en memoria represen-

tan una banda estrecha alrededor de la solución.

2.3.10.4 SMA*

SMA* usa la memoria que tenga disponible para evitar repetir estados, y es completo si su memoria es suficientemente grande para almacenar el camino de solución más corto.

Las extensiones sobre MA* son: una estructura de datos más eficiente, usa sólo 2 valores de costo (en lugar de 4), elimina sólo lo estrictamente necesario, y usa "pathmax" como valores propagados hacia atrás (la contribución principal).

2.3.10.5 Mejoras Dinámicas

Las evaluaciones en las heurísticas son básicamente estáticas e involucran cierto error (e.g., subestimaciones).

Una forma de mejorar las funciones de evaluación es actualizarlas dinámicamente conforme se obtiene más informacón.

Algunas variantes incluyen:

- *Add method*: (i) Se calcula alrededor de la meta rutas a varios puntos (B_is), (ii) Se estima el error entre el valor real de todos estos puntos ($g*(B_i)$) y el estimador inicial a estos puntos ($Diff(B_i) = g*(B_i) - h(B_i)$, (iii) se toma el error mínimo ($minDiff = min_i(Diff(B_i)$ y se usa para mejorar el estimador de los nodos (As) haciendo búsqueda hacia adelante ($newh(A) = h(A) + minDiff$).

- *Max method*: (i) Se calcula alrededor de la meta rutas a varios puntos (B_is), (ii) Se estima la distancia entre el estado inicial y los puntos cercanos a la meta y se estima el mínimo de la distancia total ($fmin2 = min_i(g*(B_i) + h(B_i))$, (iii) se usa para mejorar el estimador de los nodos (As) haciendo búsqueda hacia adelante, tomando en cuenta el estimador inicial hacia $A, h_1(A)$ ($h'(A) = fmin2 - h_1(A)$), (iv) como no

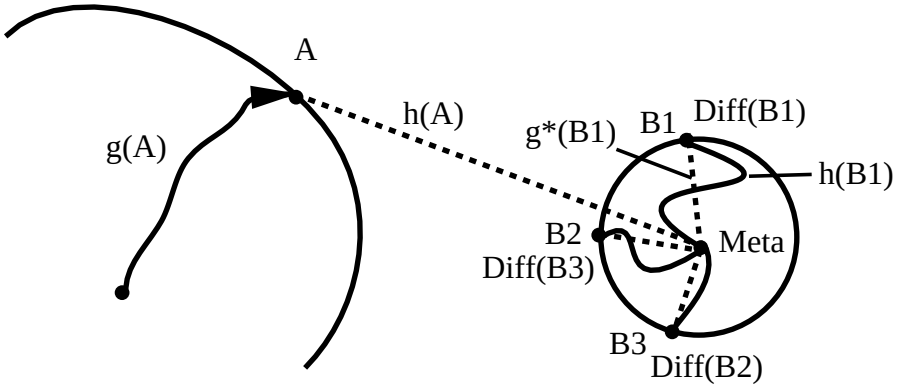

Figura 2.12: Esquema ilustrando el método *Add*

siempre es mejor estimador que el que se tenía, se hace la combinación max de ambos ($newh(A) = max(h(A), fmin2 - h_1(A))$.

- *Back-up method*: (i) Se toman los sucesores de un nodo A (Bs), (ii) Se estima el costo de A a cada uno de los Bs ($k_i(A, B_i)$) y la estimación de cada B a la meta ($h(B_i)$). (iii) Se usa como estimador de A a la meta, el mínimo de estos ($newh(A) = min_i(k_i(A, B_i) + h(B_i))$). Si el evaluador no es consistente, entonces se toma: $newh(A) = max(h(A), min_i(k_i(A, B_i) + h(B_i)))$.

- *Front-to-front method*: (i) Se calcula alrededor de la meta rutas a varios puntos (B_is), (ii) Se toma como nuevo estimador de un nodo A, el mínimo entre las estimaciones de A a cada B_i ($h(A, B_i)$) junto con el costo de cada B_i a la meta ($k_i(B_i, Meta)$), osea: $newh(A) = min_i(k_i(B_i, Meta) + h(A, B_i))$). De nuevo, si el evaluador no es consistente, entonces se toma: $newh(A) = max(h(A), min_i(k_i(B_i, Meta) + h(A, B_i)))$.

Existen muchas otras variantes que se ido proponiendo en la literatura.

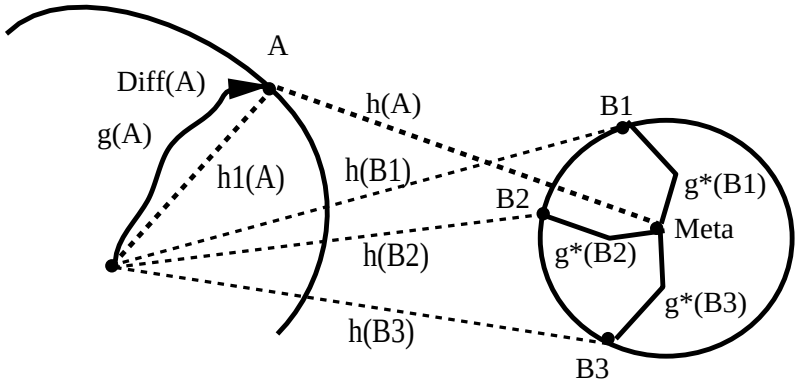

Figura 2.13: Esquema ilustrando el método *Max*

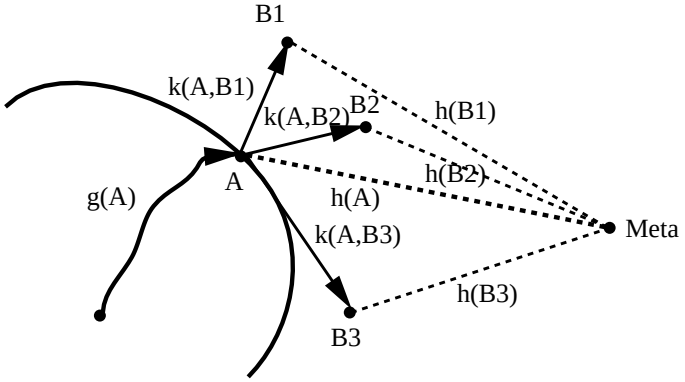

Figura 2.14: Esquema ilustrando el método *Back-up*

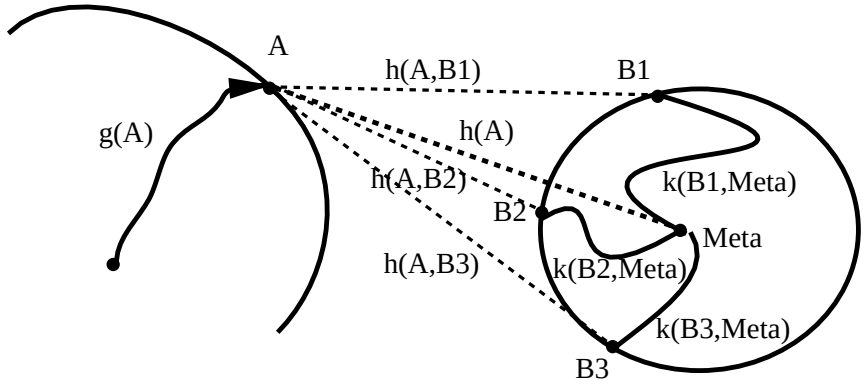

Figura 2.15: Esquema ilustrando el método *Front-to-front*

2.4 Cómo Inventar Heurísticas

Para el 8–puzzle, es más o menos fácil convencer que la heurística que cuenta las distancias Manhattan entre cuadros y sus lugares es una buena heurística.

Lo más sorprendente es que la estimación y el convencimiento es por que cumple $h(n) \leq h^*(n)$, donde $h^*(n)$ la cual desconocemos (por eso hacemos la estimación).

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.

La mayoría de las heurísticas no sólo son admisibles sino también consistentes.

Por ejemplo, podemos usar la notación de STRIPS para representar movimientos (ver tabla 2.13).

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

Tabla 2.13: La accción de mover un cuadro en el 8-puzzle usando notación de Strips.

MUEVE(CuadroX,LugarY,LugarZ):
  Precond.:
    EN(CuadroX,LugarY),
    LIBRE(CuadroZ),
    ADY(CuadroY,CuadroZ)
  Añade:
    EN(CuadroX, LugarZ),
    LIBRE(LugarY)
  Elimina:
    EN(CuadroX,LugarY),
    LIBRE(LugarZ)

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.

Si queremos añadir más posibilidades, lo que tenemos que hacer es expresar las relaciones, tales como ADJ en más simples, eg., VECINOS y MISMA–LINEA.

Eliminando una de las dos, dá nuevas heurísticas.

Por ejemplo un programa ABSOLVER se escribió para generar automáticamente heurísticas usando varios métodos. Generó nuevas heurísticas para el 8–puzzle mejores que las existentes y encontró una heurística útil para el cubo de Rubik.

Sólo hay que tener cuidado, a veces el problema resultante de eliminar restricciones, 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. Por

ejemplo, si observamos que casi siempre que nuestra heurística nos da un cierto valor, la distancia real es otra, podemos usar ese otro valor. Esto deja de garantizar la admisibilidad, pero puede dar en promedio muy buenos resultados.

Los estimadores (e.g., h en A*), puede ser un estimador estadístico y se pueden utilizar técnicas de muestreo estadístico.

Otra forma es escoger un conjunto de atributos que puedan contribuir al cálculo de la función de evaluación. Para ésto, podemos usar un algoritmo de aprendizaje que aprenda los coeficientes adecuados de estos atributos.

Hemos asumido que el costo de cálculo de una heurística es aproximadamente el mismo que el costo de expander un nodo. Pero si el costo es mucho mayor, se tiene que reconsiderar.

2.5 Satisfacción de Restricciones

Problemas de satisfacción de restricciones es un tipo especial de problemas que satisfacen algunas propiedades adicionales.

Las restricciones pueden involucrar una o varias variables al mismo tiempo.

A veces en estos problemas conviene hacer una verificación hacia adelante (*forward checking*) para detectar estados sin solución (e.g., las 8–reinas).

Muchas veces lo que conviene es analizar la variable más restringida, ésto es, asignarle un valor a la variable que está involucrada en la mayor cantidad de restricciones.

Otra heurística común es seleccionar un valor que elimine el menor número de valores en las otras variables asociadas a la variable por medio de una restricción.

A veces la descripción del estado contiene toda la información necesaria para llegar a una solución (e.g., las 8-reinas) y se utilizan algoritmos que hacen mejoras iterativas.

La idea general es empezar con una configuración completa y hacer modificaciones para mejorar su calidad.

Normalmente, en problemas de maximización se trata de moverse hacia el pico más alto. Los métodos iterativos normalmente guardan sólo su estado actual y no ven mas allá de sus vecinos inmediatos.

Aquí podemos mencionar a gradiente descendiente (hill–climbing) y a recocido simulado (pero esos los veremos más adelante).

2.6 Búsqueda en Lisp y Prolog

Las tablas 2.14 y 2.15 muestran códigos de como implementar algoritmos de búsqueda en Lisp y Prolog respectivamente.

2.7 MACRO–Operadores

Los operadores usados en planificación generalmente tienen:

- Pre–condiciones: determinan cuándo (en qué estados) puede aplicarse.
- Acciones: cambian el estado actual en un nuevo estado.

La solución de un problema implica una búsqueda de 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.

Tabla 2.14: Código en Lisp de algoritmos de búsqueda.

```
(defun busca (nodoI, nodoF)
   (busca2 (list nodoI) nodoF))

(defun busca2 (agenda nodoF)
   (cond ((null agenda) nil)
         ((equal (car agenda) nodoF))
         (t (busca2 (nva_agenda (car agenda) (cdr agenda))
                    nodoF))))
```

depth search

```
(defun nva_agenda (nodo agenda)
   (append (expande nodo) agenda))
```

breath-first

```
(defun nva_agenda (nodo agenda)
   (append agenda (expande nodo)))
```

best-first search

```
(defun nva_agenda (nodo agenda)
   (sort (append (expande nodo) agenda)))
```

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 (expande nodo) agenda))))
```

Tabla 2.15: Código en Prolog de algoritmos de búsqueda.

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

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

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).

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 | _]).
```

Si se llega a una solución (o sub–meta), guardar la secuencia de operadores que se utilizaron como un nuevo "macro"–operador.

Los macros:

- Reducen la profundidad del árbol de búsqueda al considerar secuencias largas de operadores como una sola.
- Aumentan el *branching factor*.

Strips guardaba planes generalizados en tablas triangulares.

Para generalizar un plan (y por lo tanto para que sirva en nuevas situaciones), STRIPS transforma las constantes en variables y para cada operador del plan, re–evalúa sus precondiciones (para asegurarse de no sobre–generalizar).

El uso de macros es común en la vida diaria (e.g., ir del trabajo a casa).

Korf le dió impulso a los macros ('85, '87) resolviendo problemas de manera eficiente (*8–puzzle* y *15–puzzle* y el cubo de Rubik).

Korf construye tablas de macros.

En el 8-puzzle, las 181,440 posiciones posibles, se pueden resolver sin búsqueda usando una tabla de 35 macros.

La tabla de macro operadores para el *8-puzzle* se muestra en la tabla 2.16. 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 1.2.

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.

- Permite la simplificación del problema.

Tabla 2.16: Tabla de macro operadores para el *8-puzzle*.

Pos	Cuadros			
	0	1	2	3
0				
1	UL			
2	U	RDLU		
3	UR	DLURRDLU	DLUR	
4	R	LDRURDLU	LDRU	RDLLURDRUL
5	DR	ULDRURDLDRUL	LURDLDRU	LDRULURDDLUR
6	D	URDLDRUL	ULDDRU	URDDLULDRRUL
7	DL	RULDDRUL	DRUULDRDLU	RULDRDLULDRRUL
8	L	DRUL	RULLDDRU	RDLULDRRUL

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

- Ayuda a identificar la repetición de submetas.

Las submetas pueden ser:

- Independientes: solución óptima se obtiene al concatenar soluciones óptimas de submetas.
- Serializables: existe un cierto orden de las submetas que resuelven el problema sin violar una submeta resuelta con anterioridad.
- No serializables: se necesitan violar soluciones anteriores para resolver el problema.

Los macros pueden ayudar a resolver problemas no serializables.

El beneficio de los macros decae al incrementar su número (i.e., al resolver más problemas) y es necesario usar filtros de *aplicabilidad*.

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 (relacionado con *Case–based reasoning*).

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.

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.

Problemas: ciegan la vista del solucionador de problemas (sobretodo en ambientes dinámicos).