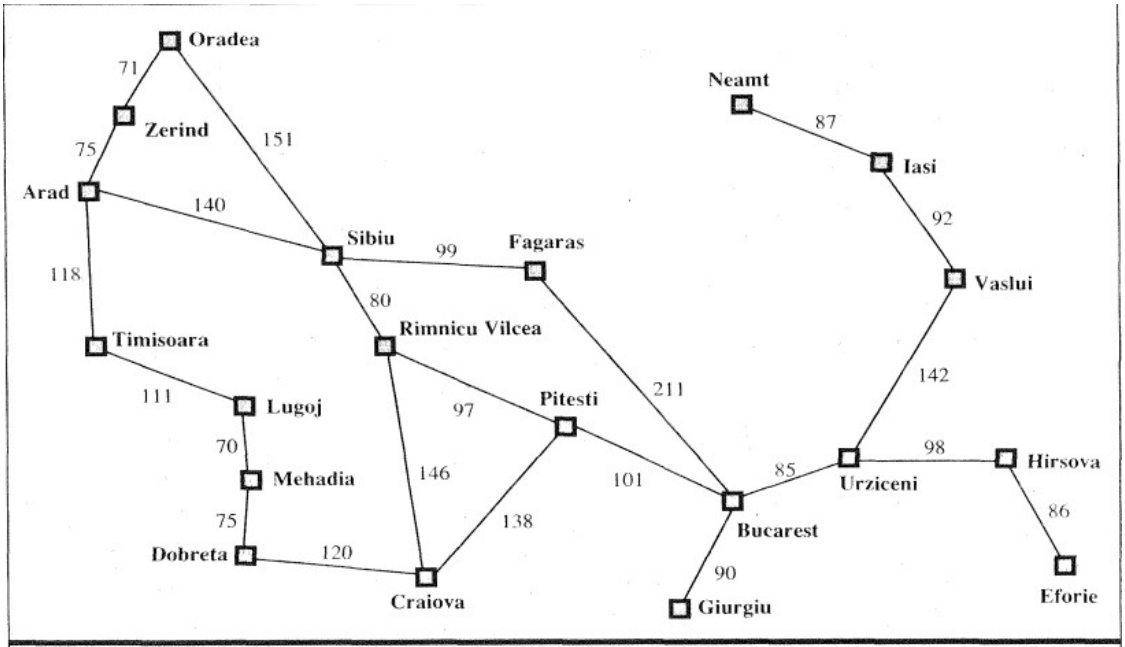

Figura 3.2 Un mapa de carreteras simplificado de parte de Rumanía.

Búsqueda de soluciones

Comenzaremos a revisar la forma de encontrar soluciones mediante un árbol de búsqueda generado a partir del estado inicial. Considere el ejemplo de la figura 3.2, queremos encontrar un camino desde Arad hasta Bucarest. La raíz del árbol de búsqueda debe ser el nodo correspondiente al estado inicial, es decir, Arad. Lo primero que se debe hacer es comprobar si el estado inicial es el estado objetivo (Bucarest), como no lo es, se debe expandir el estado actual, generando un nuevo conjunto de estados. En este caso se generan tres nuevos estados Sibiu, Timisoara y Zerind. Ahora se debe elegir cual de estos estados se va a evaluar.

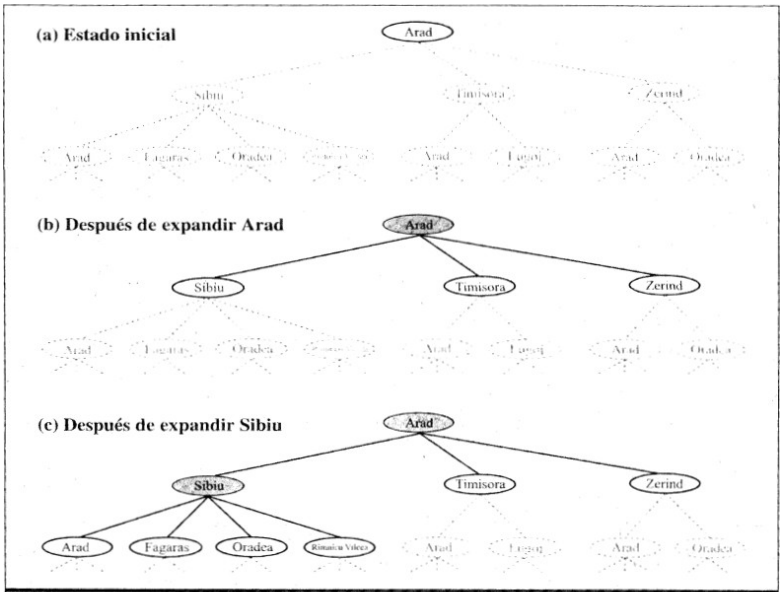

Figura 3.6 Árboles de búsqueda parciales para encontrar una ruta desde Arad hasta Bucarest. Se sombrean los nodos que se han expandido; los nodos que se han generado pero todavía no se han expandido se rodean en negrita; los nodos que todavía no se han generado se marcan con líneas discontinuas.

Esto es la esencia de la búsqueda, elegimos un camino y dejamos el resto para más tarde. El estado a expandir está determinado por la estrategia de búsqueda.

Es importante distinguir entre el espacio de estados y el árbol de búsqueda. Para este ejemplo, hay solamente 20 estados en el espacio de estados, uno por cada ciudad. Pero hay un número infinito de caminos en este espacio de estados, así que el árbol de búsqueda tiene un número infinito de nodos. Por ejemplo Arad-Sibiu-Arad-Sibiu…. Obviamente un buen algoritmo de búsquedas evitaría seguir trayectorias de este tipo.

Hay muchas formas de representar los nodos, consideremos la siguiente estructura:

- **Estado**: el estado, del espacio de estados, que corresponde con el nodo.
- **Nodo padre**: el nodo en el árbol de búsqueda que ha generado a éste nodo.
- **Acción: la acción**: que se aplicó al nodo padre para generar éste nodo.
- **Costo del camino**: el costo del camino desde el estado inicial hasta éste nodo.
- **Profundidad**: el número de pasos a lo largo del camino desde el estado inicial.
- **Frontera**: Colección de nodos que se han generado a partir de éste nodo.

La estrategia de búsqueda será una función que que seleccione de la frontera el siguiente nodo a expandir. Aunque esto suena conceptualmente sencillo, podría ser computacionalmente costoso, ya que la función quizá tenga que revisar a cada uno de los nodos e la frontera para elegir al mejor. Consideraremos a la frontera como una lista con las siguientes funciones:

- CrearLista(elemento,….) Crea una lista con los elementos dados
- EsVacia() devuelve verdadero si no hay ningún elemento en la lista
- Primero() devuelve el primer elemento de la lista
- BorrarPrimero() devuelve Primero() y lo borra de la lista
- Inserta(elemento) inserta un elemento al final de la lista

Rendimiento de la resolución del problema

La forma de evaluar el desempeño de un algoritmo de búsqueda de éste tipo es comúnmente:

- Completitud: ¿está garantizado que el algoritmo encuentre una solución cuando exista?
- Optimización: ¿encuentra la solución optima?
- Complejidad en tiempo: ‘cuánto tarda en encontrar la solución?
- Complejidad en memoria: ¿cuánta memoria se necesita para el funcionamiento de la búsqueda?

El tiempo a menudo se mide en términos del número de nodos generados durante la búsqueda, y el espacio en términos de el máximo número de nodos que se almacena en memoria.

Estrategias de búsqueda no informada

El término búsqueda no informada (búsqueda a ciegas) implica que no se posee información adicional acerca de los estados más allá de la que proporciona la definición del problema. Todo lo que pueden hacer es generar los sucesores y distinguir un estado objetivo de uno que no lo es. Las estrategias que si saben si un estado no objetivo es más prometedor que otro se llaman búsqueda informada o búsqueda heurística.

Búsqueda primero en anchura

Es una estrategia sencilla en la que se expande primero el nodo raíz, a continuación se expanden todos los sucesores del nodo raíz, después los sucesores de éstos, etc. En general, se expanden todos los nodos en la misma profundidad en el árbol de búsqueda antes de expandir cualquier nodo del próximo nivel.

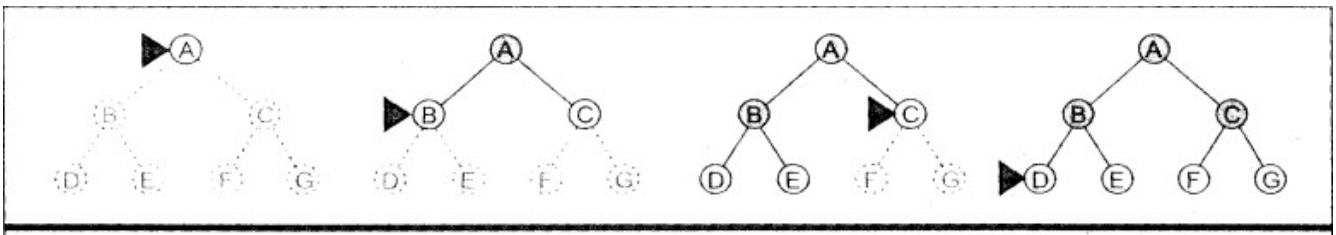

Figura 3.10 Búsqueda primero en anchura sobre un árbol binario sencillo. En cada etapa, el próximo nodo a expandir se indica con una marca.

Evaluemos la búsqueda primero en anchura. Primero podemos ver fácilmente que es completa, se expande la totalidad de nodos y en algún momento se llegara a la solución (si existe). El nodo objetivo más superficial (el primero que se encuentra con esta técnica) no es necesariamente el óptimo. La búsqueda en anchura solo es optima si el costo del camino es una función no decreciente de la profundidad del nodo, por ejemplo, cuando todas las acciones tienen el mismo costo.

Para evaluar el costo en tiempo y espacio consideremos un espacio de estados hipotetico, cada estado tiene **b** sucesores. La raíz del árbol genera **b** nodos en el primer nivel, cada uno de ellos genera **b** nodos en el siguiente nivel teniendo un total de $\mathbf{b^2}$ nodos. Cada uno genera **b** nodos más teniendo $\mathbf{b^3}$ nodos en el tercer nivel, etc. Ahora supongamos que la solución se encuentra en el nivel **d**. En el peor caso expandiremos todos excepto el último nodo del nivel **d** (ya que el nodo objetivo no se expande), generando **b^d+1 – b** nodos en el nivel **d+1**. Entonces el número total de nodos generados es

$$\mathbf{b+ b^2+b^3+…..+b\text{^}d + (b\text{^}d+1 – b) = O(b\text{^}d+1)}$$

Cada nodo debe permanecer en la memoria, porque o es parte de la frontera o es un antepasado de un nodo frontera (padre). La complejidad en espacio es, por lo tanto, la misma que la complejidad en tiempo (más el nodo raiz)

Consideremos b = 10, una velocidad para generar 1,000,000 nodos por segundo y 1000 bytes por nodo para almacenarlos en memoria.

Profundidad	Nodos	Tiempo	Memoria
2	111	0.11 miliseg	108 kilobytes
4	11111	11.11 miliseg	10.59 megabytes
6	1111111 ~ 10^6	1.111111 seg	1.03 gigabytes
8	111111111 ~ 10^8	1.85 min	103.48 gigabytes
10	1111111111 ~ 10^10	3.08 hrs	10.1 terabytes
12	1111111111111 ~ 10^12	12.86 dias	1010.5 terabytes
14	111111111111111 ~ 10^14	3.57 años	98.6 petabytes
16	1111111111111111 ~ 10^16	357.2 años	9.6 exabytes

De la tabla observamos que es un problema más grande los requisitos de memoria que el tiempo de ejecución. 3.08 horas no sería demasiado tiempo de espera para la solución de un problema importante a profundidad 8, pero pocas computadoras tienen más de 10 terabytes para almacenar el árbol.

Los requisitos de tiempo todavía son un factor importante, bajo nuestras suposiciones, una solución a profundidad 14 tardaría 3.57 años. En general, los problemas de búsqueda con complejidad exponencial no pueden ser resueltos por métodos sin información, salvo casos pequeños.

Búsqueda de costo uniforme

La búsqueda primero en anchura es óptima cuando todos los costos son iguales, porque siempre expande el nodo más superficial. Con una extensión sencilla, podemos encontrar un algoritmo que es óptimo con cualquier función costo. En lugar de expandir el nodo más superficial, vamos a expandir el nodo **n** con el camino de costo más pequeño. Notemos que si todos los caminos tienen el mismo costo, es idéntico a primero en anchura.

Por ejemplo, si queremos ir de Sibiu a Bucaarest, los sucesores de Sibiu son Rimnicu Vilcea y Fagaras, con costos de 80 y 90 respectivamente. El menor costo es el de Rimnicu Vilcea, por lo tanto se expande. Ahora debemos elegir entre expandir a Pitesti y Fagaras, Pitesti tiene un costo de 80+97 = 177 y Fagaras tiene un costo de 99, por lo tanto expandimos Fagaras. Ahora tenemos que elegir entre expandir Pitesti o Bucarest (Note que no hemos se comprueba si el sucesor es solución sino hasta ser expandido). Bucarest tiene un costo de 99+211 = 310 mientras que pitesti tiene un costo de 177, entonces expandimos Pitesti. Ahora debemos elegir si expandimos Bucarest (por el lado de pitesti) o Bucarest (por el lado de Fagaras) el primero tiene un costo de 80+97+101 = 177+101 = 278, el segundo es de 99+211 = 310, por lo tanto expandemos Bucarest (por el lado de Pitesti), al expandirlo, comprobamos que es el nodo solución y encontramos el camino más corto.

Para que la búsqueda de costo uniforme funcione, se debe garantizar que los costos sean siempre positivos, no pueden ser cero ni negativo, de ser así la búsqueda entraría en un ciclo. El costo siempre debe ir incrementándose. Si garantizamos esto entonces garantizamos la completitud. Podemos garantizar también que la solución encontrada es la óptima, pues siempre se expande el nodo con el menor costo.

El costo de tiempo y espacio para ésta búsqueda no es trivial, no depende de b y d como en el caso anterior. En su lugar C* es el costo de la solución óptima y como cada acción cuesta un incremento en el costo e. Entonces la complejidad en tiempo y espacio en el peor de los casos es O(b^(C*/e)), la cual puede ser mucho mayor que b^d. Esto porque la búsqueda de costos uniformes explora árboles grandes en pequeños pasos antes de explorar caminos que implican pasos grandes y quizás más útiles.

Búsqueda primero en profundidad

La búsqueda primero en profundidad siempre expande el nodo más profundo en la frontera actual del árbol. Ésta búsqueda tiene unos requisitos muy modestos en términos del espacio. Necesita almacenar sólo un camino desde la raíz hasta un nodo hoja, junto con los nodos hermanos no expandidos. Una vez que un nodo se ha expandido, se puede quitar de la memoria tan pronto como todos sus descendientes hayan sido explorados. Para un espacio de estados con factor b y profundidad máxima m, la búsqueda primero en profundidad requiere almacenar sólo b*m +1 nodos. Suponiendo lo mismo que antes b=10 y suponiendo que los nodos a la misma profundidad que el nodo objetivo no tienen ningún sucesor, entonces ésta búsqueda requiere sólo 157.22 kilobytes para una profundidad de 16, en lugar de 9.6 exabytes de la búsqueda en profundidad.

El inconveniente de ésta búsqueda es que puede hacer una elección equivocada y obtener un camino muy largo (incluso infinito) aún cuando una elección diferente llevaría a una solución más rápida. En el peor de los casos, la búsqueda primero en profundidad genera todos los nodos del árbol de búsqueda por lo tanto su tiempo de ejecución es de O(b^m) donde m es la profundidad máxima de cualquier nodo. Note que m puede ser infinito si el árbol es ilimitado.

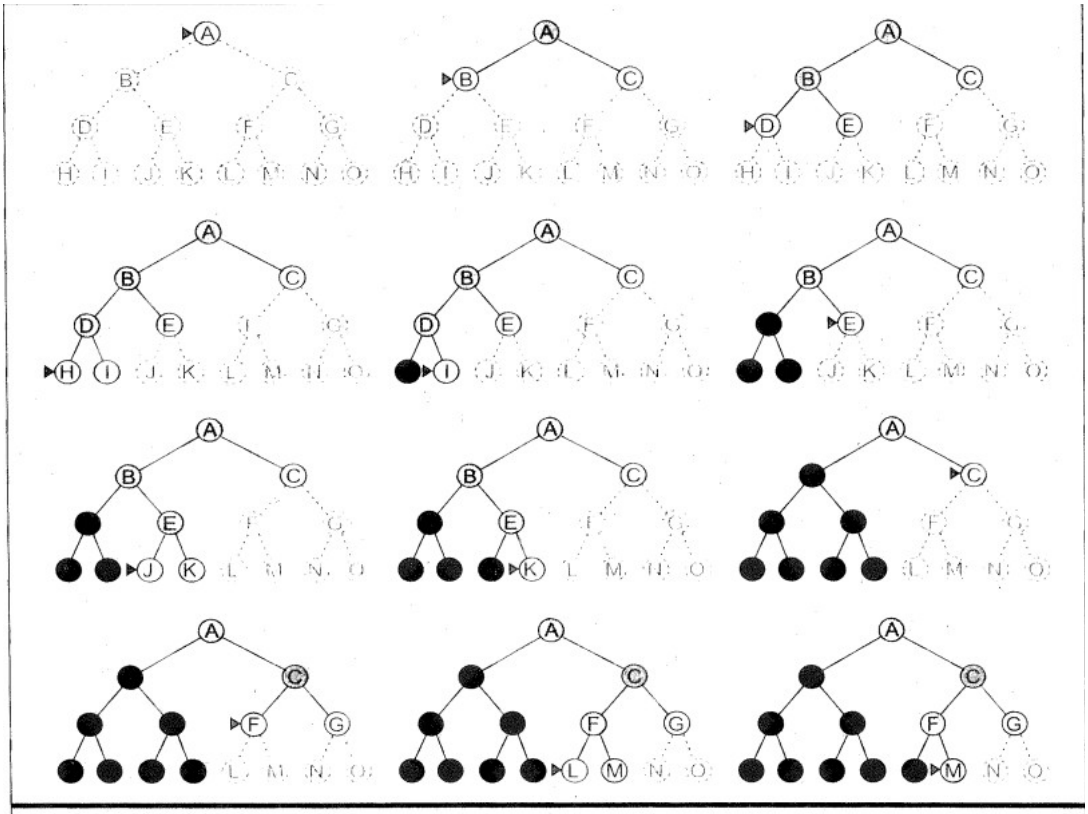

Figura 3.12 Búsqueda primero en profundidad sobre un árbol binario. Los nodos que se han expandido y no tienen descendientes en la frontera se pueden quitar de la memoria; estos se muestran en negro. Los nodos a profundidad 3 se suponen que no tienen sucesores y *M* es el nodo objetivo.

Búsqueda de profundidad limitada

Se puede resolver el problema de los árboles ilimitados aplicando la búsqueda primero en profundidad con un límite de profundidad l. Es decir, los nodos a profundidad l se tratan como si no tuvieran sucesores. El limite de la profundidad resuelve el problema del camino infinito- Lamentablemente, también introduce una fuente adicional de incompletitud si escogemos l < d, es decir, el objetivo está fuera del limite. La búsqueda de profundidad limitada también será no óptima si elegimos l > d. Su complejidad en tiempo es O(b^l) mientras que la complejidad en espacio es O(bl).

A veces los límites de profundidad pueden estar basados en el conocimiento del problema. Por ejemplo, en el mapa de Rumanía hay 20 ciudades. Por lo tanto sabemos que si hay una solución, debe ser de longitud 19 como mucho, entonces l = 19 es una opción posible. Pero de hecho si estudiamos el mapa con más cuidado, observamos quq cualquier ciudad puede alcanzarse de otra en a lo más 9 pasos (Es el diámetro de el grafo que mide la mayor distancia entre cualesquiera dos vértices). Este número nos da un mejor limite para la profundidad de la búsqueda.

Búsqueda primero en profundidad con profundidad iterativa

La búsqueda con profundidad iterativa es una estrategia usada que implementa la búsqueda primero en profundidad cambiando en cada ocasión el limite de la profundidad. Esto es, primero se realiza una búsqueda primero en profundidad con un limite de profundidad 0, si no encuentra una solución, entonces realiza otra búsqueda con profundidad 1, si no encuentra solución realiza otra búsqueda con profundidad 2, y así sucesivamente hasta encontrar la solución.

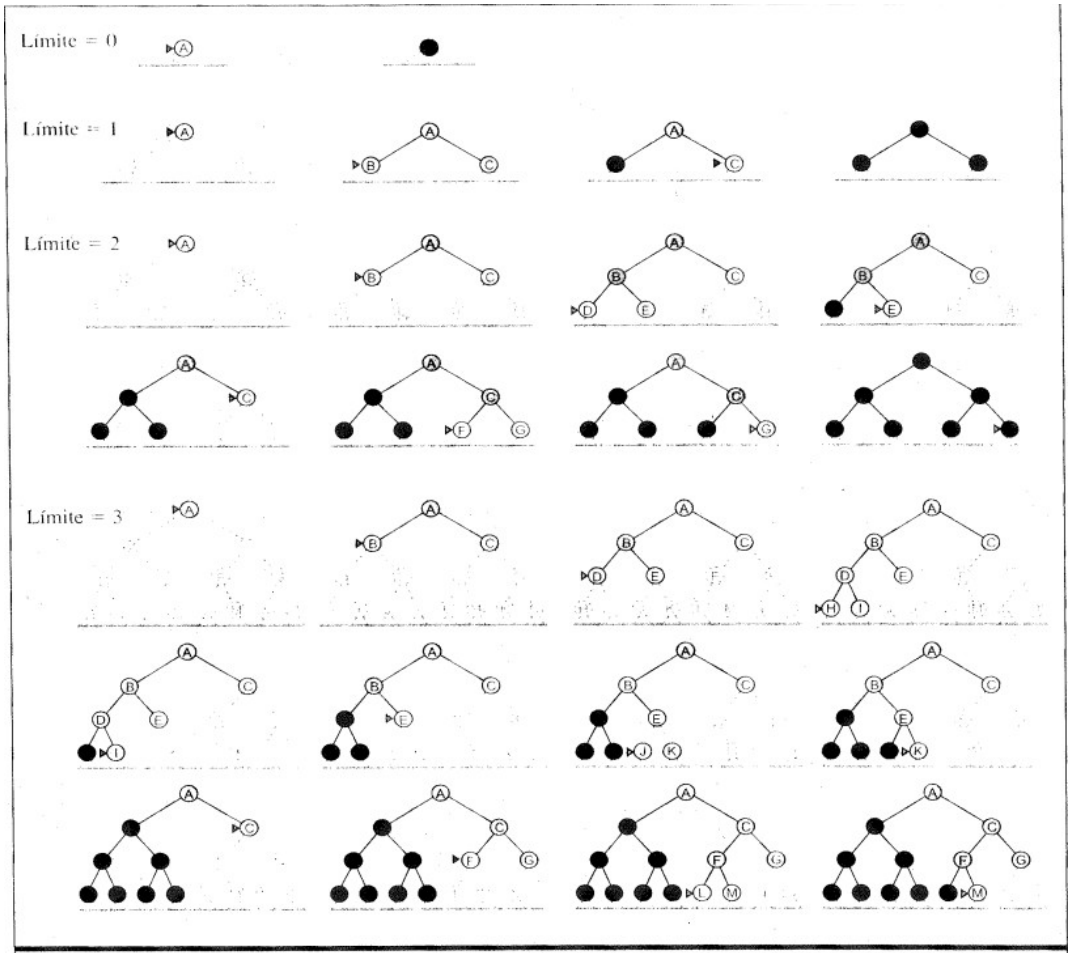

Figura 3.15 Cuatro iteraciones de la búsqueda de profundidad iterativa sobre un árbol binario.

La búsqueda de profundidad iterativa puede parecer derrochadora, porque los estados se generan múltiples veces. Pero esto no es tan costoso como parece, la razón es que en un árbol de búsqueda con el mismo factor de ramificación en cada nivel, la mayor parte de los nodos se encuentran en los niveles inferiores, entonces no importa mucho generar los niveles superiores múltiples veces. En ésta búsqueda, los nodos a profundidad d (donde esta la solución) son generados una vez, los de un nivel superior dos veces y así sucesivamente hasta los hijos de la raíz que son generados d veces. Entonces el total de nodos generados es:

$$(d)b + (d-1)b^2 + \ldots. + (1)\, b\text{^}d$$

Lo cual resulta en una complejidad O(b^d), el cual parece similar a la búsqueda primero en anchura O(b^d+1). Notemos que la búsqueda primero en anchura genera algunos nodos en profundidad d+1 (en el peor de los casos), mientras que la profundidad iterativa no lo hace. El resultado es que la profundidad iterativa es más rápida en realidad que la búsqueda primero en anchura. Por ejemplo si b=10 y d=5

N(BPI) =	50+400+3000 +20000+100000	=	123450
N(BPA) =	10+100+10000 +10000+100000+1000000	=	1111110

En general, la profundidad iterativa es el método de búsqueda no informada preferido cuando hay un espacio grande de búsqeuda y no se conoce la profundidad de la solución.

La búsqueda de profundidad iterativa es análoga a la búsqueda primero en anchura, en la cual se explora en cada iteración una capa completa de nuevos nodos antes de continuar con la siguiente capa. Parecería que vale la pena entonces desarrollar una búsqueda análoga a la búsqueda de costo uniforme, heredando las garantías de optimización del algoritmo evitando sus exigencias de memoria. La idea es usar límites crecientes en el costo en lugar de limites de profundidad.

Búsqueda bidireccional

La idea de la búsqueda bidireccional es ejecutar dos búsquedas simultaneas, una desde el estado inicial y otra desde el objetivo. Usualmente se implementa en ambos lados una búsqueda primero en anchura. La búsqueda global termina cuando ambas búsquedas se encuentran en el mismo estado intermedio. La motivación de hacer esto es que dos búsquedas con O(b^d/2) es mejor que una sola O(b^d). Por ejemplo, para b = 10 y d = 6

N(BB) = (10+100+10000) + (10+100+10000) = 2220
N(BPA) = 10+100+1000 +10000+100000+1000000 = 1111110

Comparación de las técnicas

Criterio	Primero en anchura	Costo uniforme	Primero en profundidad	Profundidad limitada	Profundidad iterativa	Bidireccional (si aplicable)
¿Completa?	Sí[a]	Sí[a, b]	No	No	Sí[a]	Sí[a, d]
Tiempo	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Espacio	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
¿Optimal?	Sí[c]	Sí	No	No	Sí[c]	Sí[c, d]

Figura 3.17 Evaluación de estrategias de búsqueda. *b* es el factor de ramificación; *d* es la profundidad de la solución más superficial; *m* es la máxima profundidad del árbol de búsqueda; ℓ es el límite de profundidad. Los superíndice significan lo siguiente: [a] completa si *b* es finita; [b] completa si los costos son $\geq \epsilon$ para ϵ positivo; [c] optimal si los costos son iguales; [d] si en ambas direcciones se utiliza la búsqueda primero en anchura.