Lectura

Inteligencia Artificial

Texto basado en las notas de Wolfgang Ertel dadas en el libro Introduction to Artificial Intelligence de Springer.

---

1 Búsqueda en Inteligencia Artificial

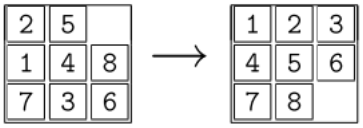
1.1 Introducción

La búsqueda de una solución en un árbol de búsqueda extremadamente grande presenta un problema para casi todos los sistemas de inferencia. Desde el estado inicial hay muchas posibilidades para el primer paso de inferencia. Para cada una de estas posibilidades, nuevamente hay muchas posibilidades en el siguiente paso y así sucesivamente. Incluso en la prueba de una fórmula muy simple con tres cláusulas de Horn, cada una con un máximo de tres literales, el tiempo total de cálculo para todas las 7.4×10^{73} inferencias sería aproximadamente igual a 1043 veces más la edad de nuestro universo, incluso contando con $10,000$ computadoras cada una de las cuales puede realizar mil millones de inferencias por segundo y además que se puede distribuir el trabajo entre todas las computadoras sin costo alguno.

Para poder dar una solución aproximada al problema de que no existe una posibilidad realista de buscar un tipo de espacio de búsqueda con los medios disponibles, primero debemos entender cómo funciona la búsqueda no uniformizada, es decir probar ciegamente todas las posibilidades. Comenzamos con algunos ejemplos.

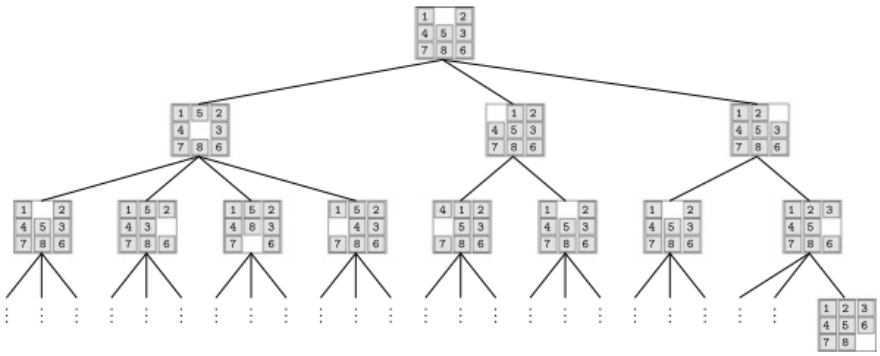
Ejemplo 1

Con el 8-puzzle, un ejemplo clásico de algoritmos de búsqueda, los distintos algoritmos se pueden ilustrar de forma muy visible. Los cuadrados con los números del 1 al 8 se distribuyen en una matriz de 3×3 como en la siguiente figura:

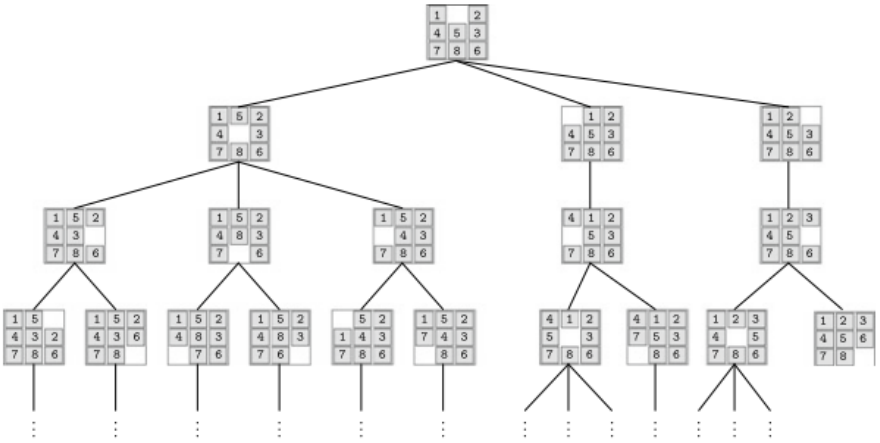


El objetivo es llegar a un cierto orden de los cuadrados, por ejemplo, en orden ascendente por filas, como se representa en la figura. En cada paso, un cuadrado se puede mover hacia la izquierda, derecha, arriba o abajo en el espacio vacío. Por tanto, el espacio vacío se mueve en la dirección opuesta correspondiente. Para el análisis del espacio de búsqueda, es conveniente mirar siempre los posibles movimientos del campo vacío.

El árbol de búsqueda de un estado inicial se representa en la siguiente figura:



Del gráfico anterior, el factor de ramificación [1] alterna entre dos, tres y cuatro. Promediado sobre dos niveles a la vez, obtenemos un factor de ramificación promedio (el factor de ramificación promedio de un árbol es el factor de ramificación que tendría un árbol con un factor de ramificación constante, igual profundidad y a la misma cantidad de nodos de hojas.) de $\sqrt{8} \approx 2.83$. También cada estado se repite varias veces dos niveles más profundo pues en una simple búsqueda no uniformizada, cada acción se puede revertir en el siguiente paso. Si no permitimos ciclos de longitud 2, entonces para el mismo estado inicial obtenemos el árbol de búsqueda representado en la siguiente figura y el factor de ramificación promedio se reduce en aproximadamente 1 y se convierte en 1.8.



Antes de comenzar a describir los algoritmos de búsqueda, se necesitan algunos términos nuevos. Estamos tratando con problemas de búsqueda discretos. Al estar en el estado s, una acción a_1 conduce a un nuevo estado $s^{'}$. Así $s^{'} = a_1(s)$. Una acción diferente puede llevar al estado $s^{''}$, en otras palabras: $s^{''} = a_2(s)$.

La aplicación recursiva de todas las acciones posibles a todos los estados, comenzando con el estado inicial, produce el árbol de búsqueda.

1.2 Problema de búsqueda

Un problema de búsqueda se define por los siguientes valores

Estado: descripción del estado del mundo en el que se encuentra el agente de búsqueda.

Estado inicial: el estado inicial en el que se inicia el agente de búsqueda.

Estado de objetivo: si el agente alcanza un estado de objetivo, finaliza y genera una solución.

Acciones: todos los agentes permitieron acciones.

Solución: la ruta en el árbol de búsqueda desde el estado inicial hasta el estado objetivo.

[1] https://stackoverflow.com/questions/47789400/how-to-find-the-branching-factor-of-a-tree

Función de costo: asigna un valor de costo a cada acción. Es necesario para encontrar una solución óptima.

Espacio de estados: el conjunto de todos los estados.

Aplicado al sistema de 8-puzzle, obtenemos:

Estado: Una matriz S 3×3 matriz con los valores $1, 2, 3, 4, 5, 6, 7, 8$ (una vez cada uno) y un cuadrado vacío.

Estado inicial: un estado arbitrario.

Estado objetivo: un estado arbitrario, por ejemplo el estado dado a la derecha del gráfico del 8-puzzle.

Acciones: Movimiento del cuadrado vacío S_{ij} a la izquierda (si $j \neq 1$), derecha (si $j \neq 3$), arriba (si $i \neq 1$), abajo (si $i \neq 3$).

Función de costo: La función constante 1, ya que todas las acciones tienen el mismo costo.

Espacio de estados: el espacio de estados está degenerado en dominios que son mutuamente inalcanzables. Por lo tanto, hay problemas de 8-puzzle sin solución.

Para el análisis de los algoritmos de búsqueda se necesitan los siguientes términos:

- El número de estados sucesores de un estado s se denomina factor de ramificación $b(s)$ o b si el factor de ramificación es constante.
- El factor de ramificación efectivo de un árbol de profundidad d con n nodos se define como el factor de ramificación que tendría un árbol con factor de ramificación constante, igual profundidad e igual a n.
- Un algoritmo de búsqueda se llama completo si encuentra una solución para cada problema resoluble. Si un algoritmo de búsqueda completo termina sin encontrar una solución, entonces el problema no tiene solución.

Para una profundidad d dada y con un conteo de nodos n, el factor de ramificación efectivo se puede calcular resolviendo la ecuación:

$$n = \frac{b^{d+1} - 1}{b - 1}$$

para b porque un árbol con factor de ramificación constante y de profundidad d tiene un total de:

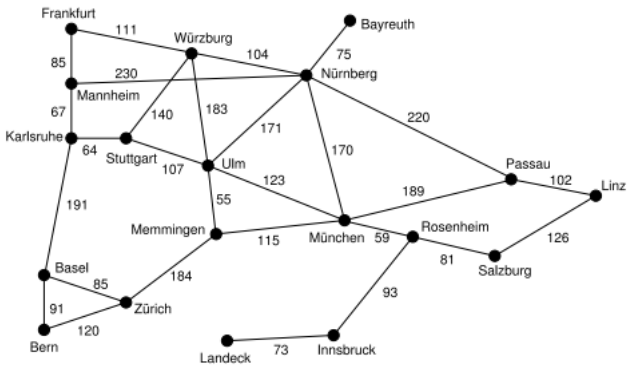
$$n = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

nodos.

Para aplicaciones prácticas de algoritmos de búsqueda de árboles de búsqueda finitos, el último nivel es especialmente importante por el siguiente:

Teorema Para árboles de búsqueda finitos muy ramificados con un factor de ramificación constante grande, casi todos los nodos están en el último nivel.

Ejemplo
Se nos da un mapa, como el que se representa en la figura, como un grafo con ciudades como nodos y conexiones de carreteras entre las ciudades como nodos ponderados con distancias. Estamos buscando una ruta óptima de la ciudad A a la ciudad B.



La descripción del esquema correspondiente dice.

Estado: una ciudad como la ubicación actual del viajero.

Estado inicial: una ciudad arbitraria.

Estado objetivo: una ciudad arbitraria.

Acciones: viajar desde la ciudad actual a una ciudad vecina.

Función de costo: la distancia entre las ciudades. Cada acción corresponde a un nodo en el grado con una distancia como peso.

Espacio de estados: todas las ciudades, es decir, nodos del grafo.

Para encontrar la ruta con una longitud mínima hay que tener en cuenta los costes porque no son constantes como en el problema del 8-puzzle.

Definición: Un algoritmo de búsqueda se llama óptimo si, si existe una solución, siempre encuentra la solución con el costo más bajo.

El problema de los 8-puzzle es determinista, lo que significa que cada acción conduce de un estado a un estado sucesor único. Además, es observable, es decir, el agente siempre sabe en qué estado se encuentra. En la planificación de rutas en aplicaciones reales no siempre se dan ambas características ¿por qué?.

Por tanto aquí, sólo veremos problemas deterministas y observables. Problemas como el 8-puzzle, que son deterministas y observables, hacen que la planificación de acciones sea relativamente simple porque, al tener un modelo abstracto, es posible encontrar secuencias de acción para la solución del problema sin realmente realizar las acciones en el mundo real. En el caso el 8-puzzle, no es necesario mover los cuadrados en el mundo real para encontrar la solución. Podemos encontrar soluciones óptimas con los denominados algoritmos offline. Uno enfrenta desafíos muy diferentes cuando, por ejemplo, se construyen robots que se supone juegan al fútbol. Aquí, nunca habrá un modelo abstracto exacto de las acciones. Por ejemplo, un robot que patea la pelota en una dirección específica no puede predecir con certeza dónde se moverá la pelota porque, entre otras cosas, no sabe si un oponente atrapará o desviará la pelota. Aquí se necesitan entonces algoritmos online, que toman decisiones basadas en señales de sensores en cada situación.

El aprendizaje por refuerzo, trabaja hacia la optimización de estas decisiones basadas en la experiencia.

2 Búsqueda no uniformizada

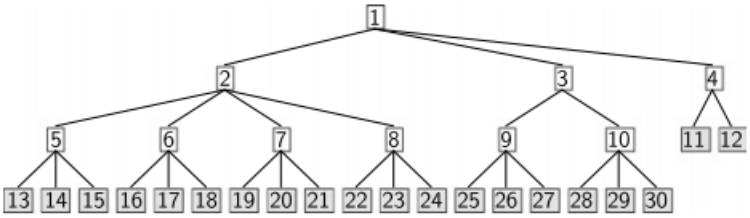
2.1 BFS (Breadth-First Search)

En un BFS, el árbol de búsqueda se explora de arriba a abajo de acuerdo con el algoritmo que se muestra a continuación hasta que se encuentra una solución. Primero, cada nodo de la lista de nodos se prueba para determinar si es un nodo objetivo y en caso de éxito, el programa se detiene. De lo contrario, se generan todos los sucesores del nodo. A continuación, la búsqueda continúa de forma recursiva en la lista de todos los nodos recién generados.

Todo se repite hasta que no se generan más sucesores. Este algoritmo es genérico. Es decir, funciona para aplicaciones arbitrarias si se proporcionan las dos funciones específicas de la aplicación `GoalReached` y `Successors`.

```
BFS(ListaNodos, Objetivo)
  NuevosNodos = vacio
For all Nodo en  ListaNodos
  If GoalReached(Nodo, Objetivo)
    Return("Solution encontrada", Nodo)
  NuevosNodos = Append(NewNodos, Successors(Nodo))
If NuevoNodos = vacio
  Return(BSD(NuevoNodos, Objetivo))
Else
  Return("No hay solucion")
```

`GoalReached` calcula si el argumento es un nodo objetivo y `Successors` calcula la lista de todos los nodos sucesores de su argumento. La figura siguiente muestra una instantánea del BFS.



Análisis

Dado que el BFS busca a través de cada profundidad del árbol y alcanza cada profundidad en un tiempo finito, es completo si el factor de ramificación b es finito. La solución óptima se encuentra si los costos de todas las acciones son los mismos. El tiempo de cálculo y el espacio de memoria crecen exponencialmente con la profundidad del árbol. Para un árbol con factor de ramificación constante b y profundidad d, el tiempo total de cálculo viene dado por:

$$c \cdot \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d).$$

Aunque solo el último nivel se guarda en la memoria, el requisito de espacio de memoria también es $O(b^d)$.

Con la velocidad de las computadoras actuales, que pueden generar miles de millones de nodos en minutos, la memoria principal se llena rápidamente y la búsqueda finaliza. El problema de que no siempre se encuentre la solución más corta puede resolverse mediante la llamada Búsqueda de Costo Uniforme, en la que el nodo con el costo más bajo de la lista de nodos (que está ordenado de forma ascendente por costo) siempre se expande y los nuevos nodos son ordenados. Así encontramos la solución óptima. Sin embargo, el problema de la memoria aún no está resuelto.

El DFS proporciona una solución a este problema.

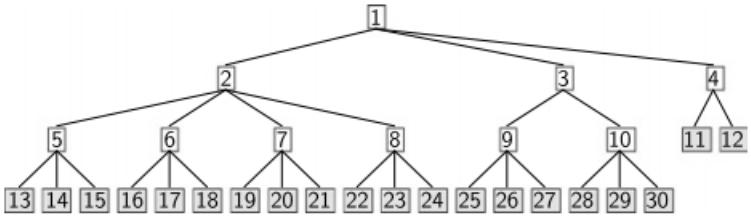
2.2 DFS

En el algoritmo DFS, solo unos pocos nodos se almacenan en la memoria a la vez. Después de la expansión de un nodo, solo se guardan sus sucesores y el primer nodo sucesor se expande inmediatamente. Así, la búsqueda rápidamente se vuelve muy profunda. Solo cuando un nodo no tiene sucesores y la búsqueda falla a esa profundidad, el siguiente nodo se expande mediante el `bactracking` a la última rama y así sucesivamente.

Podemos percibirlo mejor en el algoritmo recursivo y con el árbol de búsqueda siguientes:

```
DFS(Nodos, Objetivo)
If GoalReached(Nodo, Objetivo)
    Return("Solution encontrada")
NuevosNodos = Successors(Nodo)
While NuevoNodos = vacio
  Resultado = DFS(First(NuevoNodos), Objetivo))
  If Resultado = "Solucion encontrada" Return("Solucion encontrada")
  NuevosNodos = Rest(NuevosNodos)
Return("No hay solucion")
```

¿Qué hacen las funciones `First` y `Rest` en el algoritmo anterior?.



Análisis

El DFS requiere mucha menos memoria que el BFS porque como máximo se guardan b nodos en cada profundidad. Por lo tanto, necesitamos como máximo $b \cdot d$ celdas de memoria.

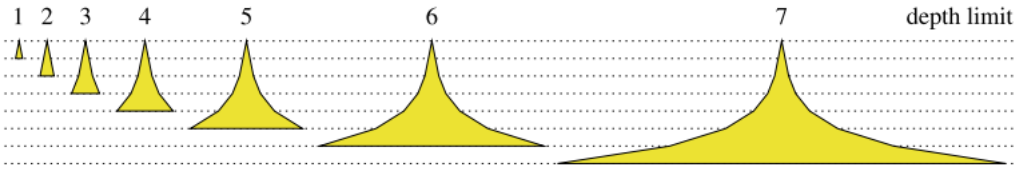
Sin embargo, el DFS no es completo para árboles infinitamente profundos porque la búsqueda de profundidad primero se ejecuta en un bucle infinito cuando no hay solución en la rama izquierda. Por tanto, la cuestión de encontrar la solución óptima es obsoleta. Debido al bucle infinito, no se puede dar ningún límite en el tiempo de cálculo. En el caso de un árbol de búsqueda de profundidad finita con profundidad d, se genera un total de aproximadamente b^d nodos. Por lo tanto, el tiempo de cálculo aumenta, al igual que el BFS, exponencialmente con la profundidad.

Podemos hacer que el árbol de búsqueda sea finito estableciendo un límite de profundidad. Ahora bien, si no se encuentra una solución en el árbol de búsqueda podado, no obstante, puede haber soluciones fuera del límite.

Por tanto, la búsqueda se vuelve incompleta. Sin embargo, existen nuevas ideas para completar la búsqueda.

2.3 Profundización iterativa

Comenzamos el DFS con un límite de profundidad de 1. Si no se encuentra una solución, aumentamos el límite en 1 y comenzamos a buscar desde el principio y así sucesivamente, como se muestra en la siguiente figura:



Esta elevación iterativa del límite de profundidad se denomina profundización iterativa. Debemos aumentar el algoritmo DFS con dos parámetros adicionales `Depth` y `Limit`. `Depth` aumenta en uno en la llamada recursiva y la línea de cabecera del bucle `while` se reemplaza por While NewNodes $\neq$ And Depth < límite. El algoritmo modificado se representa así:

```
ProfundizacionIterativa(Nodo, Objetivo)
```

```
LimiteProfundo = 0
Repeat
  Resultado = DFS-B(Nodo, Objetivo, 0, LimiteProfundo)
  LimiteProfundo = LimiteProfundo + 1
Until Resultado = "Solucion encontrada"

DFS-B(Nodos, Objetivo, Depth, Limit)
If GoalReached(Nodo, Objetivo) Return("Solution encontrada")
NuevosNodos = Successors(Nodo)
While NuevoNodos = vacio And Depth < Limit
  Resultado = DFS-B(First(NuevoNodos), Objetivo, Depth +1, Limit)
  If Resultado = "Solucion encontrada" Return("Solucion encontrada")
  NuevosNodos = Rest(NuevosNodos)
Return("No hay solucion")
```

Análisis

El requisito de memoria es el mismo que en el DFS. Se podría argumentar que reiniciar repetidamente el DFS en profundidad cero provoca mucho trabajo redundante. Para factores de ramificación grandes, este no es el caso. Se muestra que la suma del número de nodos de todas las profundidades hasta el anterior al último $d_{max} - 1$ en todos los árboles buscados es mucho menor que el número de nodos en el último árbol buscado.

Sea $N_b(d)$ el número de nodos de un árbol de búsqueda con factor de ramificación b, profundidad d y d_max la última profundidad buscada. El último árbol buscado contiene:

$$N_b(d_{max}) = \sum_{i=0}^{d_{max}} b^i = \frac{b^{d_{max}+1} - 1}{b - 1}$$

nodos. Todos los árboles juntos registrados de antemano tienen:

$$\begin{aligned}
\sum_{d=1}^{d_{max}-1} N_b(d) &= \sum_{d=1}^{d_{max}-1} \frac{b^{d+1} - 1}{b - 1} = \frac{1}{b-1}\left(\left(\sum_{d=1}^{d_{max}-1} b^{d+1}\right) - d_{max} + 1\right) \\
&= \frac{1}{b-1}\left(\left(\sum_{d=2}^{d_{max}} b^{d}\right) - d_{max} + 1\right) \\
&= \frac{1}{b-1}\left(\frac{b^{d_{max}+1} - 1}{b-1} - 1 - b - d_{max} + 1\right) \\
&\approx \frac{1}{b-1}\left(\frac{b^{d_{max}+1} - 1}{b-1}\right) = \frac{1}{b-1} N_b(d_{max}).
\end{aligned}$$

nodos. Para $b > 2$, este resultado es menor que el número $N_b(d_{max})$ de nodos en el último árbol. Para $b = 20$, los primeros $d_{max} - 1$ árboles juntos contienen solo alrededor de $\frac{1}{b-1} = 1/19$ del número de nodos en el último árbol. El tiempo de cálculo para todas las iteraciones además de la última puede ignorarse.

Al igual que el BFS, este método es completo y dado un costo constante para todas las acciones, encuentra la solución más óptima.

De los algoritmos de búsqueda descritos, el método de la profundización iterativa es el único prácticamente utilizable.

2.4 Comprobación de un ciclo

Como se ha mencionado anteriormente, los nodos pueden visitarse repetidamente durante una búsqueda. En el 8-puzzle, por ejemplo, cada movimiento se puede deshacer inmediatamente, lo que conduce a ciclos

innecesarios de longitud dos. Estos ciclos pueden evitarse registrando dentro de cada nodo todos sus predecesores y al expandir un nodo, comparar los nodos sucesores recién creados con los nodos predecesores. Todos los ciclos duplicados encontrados se pueden eliminar de la lista de nodos sucesores.

Esta simple verificación cuesta solo un pequeño factor constante de espacio de memoria adicional y aumenta el tiempo de cálculo constante c en una constante adicional δ para la verificación misma para un total de $c + \delta$. Esta sobrecarga para la verificación del ciclo se compensa con una reducción en el costo de la búsqueda. La reducción depende de la aplicación particular y por lo tanto, no se puede dar en términos generales.

Para el 8-puzzle obtenemos el resultado de la siguiente manera. Si, por ejemplo, durante el BFS con un factor de ramificación efectivo b en un árbol finito de profundidad d, el tiempo de cálculo sin la verificación del ciclo es $c \cdot b^d$, el tiempo requerido con la verificación del ciclo se convierte en:

$$(c + \delta) \cdot (b - 1)^d$$

Por tanto, la comprobación prácticamente siempre da como resultado una ganancia clara porque la reducción del factor de ramificación en uno tiene un efecto de crecimiento exponencial a medida que aumenta la profundidad, mientras que el tiempo de cálculo adicional δ solo aumenta un poco el factor constante.

Ahora surge la pregunta de cómo una verificación de ciclos de duración arbitraria afectaría el rendimiento de la búsqueda. La lista de todos los predecesores ahora debe almacenarse para cada nodo, lo cual se puede hacer de manera muy eficiente. Durante la búsqueda, cada nodo recién creado ahora debe compararse con todos sus predecesores. El tiempo de cálculo en el DFS o BFS viene dado por:

$$c_1 \cdot \sum_{i=0}^{d} b^i + c_2 \cdot \sum_{i=0}^{d} i \cdot b^i.$$

Aquí, el primer término es el costo ya conocido de generar los nodos y el segundo término es el costo de la verificación del ciclo. Podemos demostrar que para valores grandes de b y d se cumple lo siguiente:

$$\sum_{i=0}^{d} i \cdot b^i \approx d \cdot b^d.$$

Por lo tanto, la complejidad de la búsqueda con la verificación de ciclo completo solo aumenta en un factor d más rápido que para la búsqueda sin verificación de ciclo. En árboles de búsqueda que no son muy profundos, esta complejidad adicional no es importante. Para tareas de búsqueda con árboles muy profundos y de ramificación débil, puede resultar ventajoso utilizar una tabla hash para almacenar la lista de predecesores. Las búsquedas en la tabla se pueden realizar en tiempo constante, de modo que el tiempo de cálculo del algoritmo de búsqueda solo crece en un pequeño factor constante.

En resumen, podemos concluir que la verificación del ciclo apenas implica una sobrecarga adicional y por lo tanto, vale la pena para aplicaciones con nodos que ocurren repetidamente.