Lectura

Inteligencia Artificial

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

---

1 Búsqueda heurística

Las heurísticas son estrategias de resolución de problemas que, en muchos casos, encuentran una solución más rápida que una búsqueda no uniformizada. Sin embargo, esto no está garantizado pues la búsqueda heurística puede requerir mucho más tiempo e incluso puede resultar en que no se encuentre una solución.

Los humanos usamos con éxito los procesos heurísticos para todo tipo de cosas. Cuando compramos verduras en un supermercado, por ejemplo, juzgamos las diversas opciones para un kilo de fresas utilizando solo unos pocos criterios simples como el precio, la apariencia, la fuente de producción y la confianza en el vendedor y luego decidimos cuál es la mejor opción. En teoría, podría ser mejor someter las fresas a un análisis químico básico antes de decidir si comprarlas. Por ejemplo, las fresas pueden estar envenenadas. Si ese fuera el caso, el análisis habría valido la pena. Sin embargo, no llevamos a cabo este tipo de análisis porque existe una probabilidad muy alta de que nuestra selección heurística tenga éxito y nos lleve rápidamente a nuestro objetivo de comer deliciosas fresas.

Las decisiones heurísticas están estrechamente ligadas a la necesidad de tomar decisiones en tiempo real con recursos limitados. En la práctica, se prefiere una buena solución que se encuentra rápidamente a una solución que es óptima, pero muy costosa de obtener.

Se utiliza una función de evaluación heurística $f(s)$ para estados para modelar matemáticamente una heurística. El objetivo es encontrar, una solución al problema de búsqueda indicado con un costo total mínimo. Ten en cuenta que existe una sutil diferencia entre el esfuerzo por encontrar una solución y el costo total de esta solución.

A continuación, modificaremos el algoritmo BFS agregando una función de evaluación. Los nodos actualmente abiertos ya no se expanden de izquierda a derecha por fila, sino de acuerdo con su calificación heurística. Del conjunto de nodos abiertos, el nodo con la calificación mínima siempre se expande primero. Esto se logra evaluando inmediatamente los nodos a medida que se expanden y clasificándolos en la lista de nodos abiertos. Entonces, la lista puede contener nodos de diferentes profundidades en el árbol.

Debido a que la evaluación heurística de estados es muy importante para la búsqueda, a partir de ahora diferenciaremos entre estados y sus nodos asociados. El nodo contiene el estado e información relevante para la búsqueda, como su profundidad en el árbol de búsqueda y la calificación heurística del estado. Como resultado, la función `Successors` que genera los sucesores (hijos) de un nodo, también debe calcular inmediatamente para estos nodos sucesores sus calificaciones heurísticas como un componente de cada nodo.

Definimos el algoritmo de búsqueda general `HEURISTICSEARCH`:

```
HEURISTICSEARCH(Inicio, Objetivo)
```

```
ListaNodos = [Inicio]
While True
  If ListaNodos = vacio  Return("No hay solucion")
  Nodo = First(ListaNodos)
  ListaNodos = Rest(ListaNodos)
  If GoalReached(Nodo, Objetivo)Return("Solution encontrada", Nodo)
  ListaNodos = SortIn(Successors(Nodo),ListaNodos)
```

La lista de nodos se inicializa con los nodos iniciales. Luego, en el ciclo se elimina el primer nodo de la lista y se prueba si es un nodo solución. En caso contrario, se amplia con la función `Successors` y sus sucesores se añaden a la lista con la función `SortIn`. `SortIn (X, Y)` inserta los elementos de la lista X sin clasificar en la lista Y ordenada de forma ascendente. La clasificación heurística se utiliza como clave de clasificación. Por lo tanto, se garantiza que el mejor nodo (es decir, el que tiene el valor heurístico más bajo) está siempre al principio de la lista.

Los algoritmos DFS y BFS son casos especiales de la función `HEURISTICSEARCH`. Podemos generarlos fácilmente conectando la función de evaluación adecuada (¿cómo?).

La mejor heurística sería una función que calcula los costos reales desde cada nodo hasta el objetivo. Sin embargo, para hacer eso, se requiere recorrer todo el espacio de búsqueda, que es exactamente lo que se supone que debe evitar la heurística. Por tanto, necesitamos una heurística que sea rápida y sencilla de calcular. ¿Cómo encontramos tal heurística?

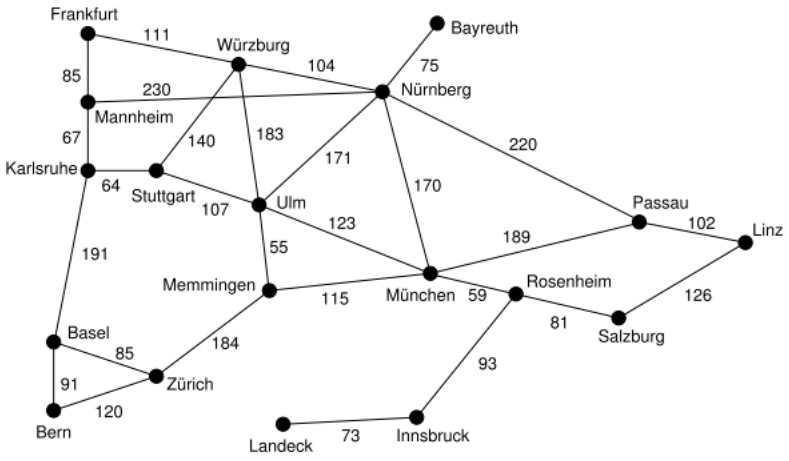
Una idea interesante para encontrar una heurística es la `simplificación del problema`. La tarea original simplificada como para que pueda resolverse con un costo computacional mínimo. Los costos de un estado al objetivo en el problema simplificado sirven entonces como una estimación del problema real. Esta función de estimación de costos la denotamos h.

1.1 Búsqueda codiciosa

Elegir el estado con el valor h estimado más bajo (es decir, el que tiene el costo estimado más bajo) de la lista de estados disponibles actualmente parece sensato. La estimación de costes se puede utilizar directamente como función de evaluación. Para la evaluación en la función `HEURISTICSEARCH` establecemos $f(s) = h(s)$. Esto se puede ver claramente en el ejemplo de planificación de viajes de la nota anterior. Establecemos la tarea de encontrar la ruta en línea recta de una ciudad a otra (es decir, la distancia de vuelo) como una simplificación del problema. En lugar de buscar la ruta óptima, primero determinamos de cada nodo una ruta con una distancia mínima de vuelo hasta el objetivo. Elegimos Ulm como destino. Por lo tanto, la función de estimación de costos se convierte en:

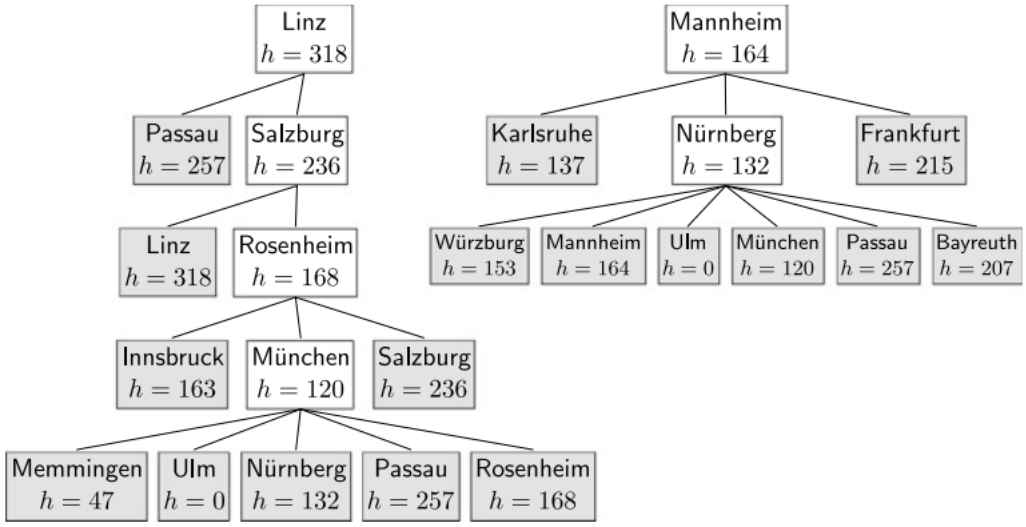
$$h(s) = \text{distancia de vuelo desde la ciudad s hasta Ulm.}$$

Las distancias de vuelo desde todas las ciudades a Ulm se muestran en la figura junto al grafo:



Basel	204
Bayreuth	207
Bern	247
Frankfurt	215
Innsbruck	163
Karlsruhe	137
Landeck	143
Linz	318
München	120
Mannheim	164
Memmingen	47
Nürnberg	132
Passau	257
Rosenheim	168
Stuttgart	75
Salzburg	236
Würzburg	153
Zürich	157

El árbol de búsqueda para comenzar en Linz hasta Ulm (izquierda) y de Mannheim a Ulm (derecha) se muestra en el siguiente gráfico:



Node	München	Innsbruck	Salzburg	Passau	Linz
heuristic rating	120	163	236	257	318

Podemos ver que el árbol es muy delgado. Por tanto, la búsqueda termina rápidamente. Desafortunadamente, esta búsqueda no siempre encuentra la solución óptima. Por ejemplo, este algoritmo no encuentra la solución óptima al comenzar en Mannheim. El camino Mannheim-Nuremberg-Ulm tiene una longitud de 401 km. La ruta Mannheim-Karlsruhe-Stuttgart-Ulm sería significativamente más corta con 238 km. Al observar el grafo, la causa de este problema se vuelve clara. De hecho, Nurnberg está algo más cerca que Karlsruhe de Ulm, pero la distancia de Mannheim a Nurnberg es significativamente mayor que la de Mannheim a Karlsruhe. La heurística solo mira hacia adelante `con codicia` hacia el objetivo en lugar de tener en cuenta también el tramo que ya se ha establecido para el nodo actual. Eso se llama búsqueda codiciosa.

1.2 Búsqueda A^*

Ahora queremos tener en cuenta los costos que se han acumulado durante la búsqueda hasta los nodos actuales. Primero definimos la función de costo:

$$g(s) = \text{Suma de costos acumulados desde el inicio hasta el nodo actual,}$$

luego sea agrega el costo estimado para el objetivo y se obtiene como función de evaluación heurística:

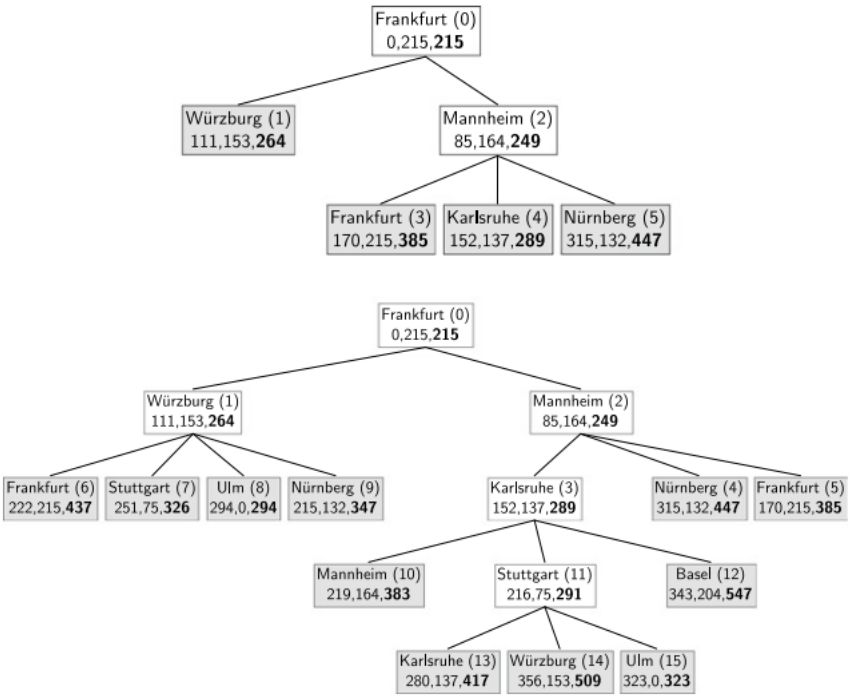
$$f(s) = g(s) + h(s)$$

Ahora agregamos otro requisito pequeño, pero importante.

Definición Una función heurística de estimación de costos $h(s)$ que nunca sobreestima el costo real del estado s al objetivo se denomina admisible.

La función `HEURISTICSEARCH` junto con una función de evaluación $f(s) = g(s) + h(s)$ y una función heurística admisible h se llama algoritmo A^*. Este famoso algoritmo es completo y óptimo. Por lo tanto A^* siempre encuentra la solución más corta para cada problema de búsqueda con solución.

Aplicamos el algoritmo A^* al ejemplo. Buscamos el camino más corto de Frankfurt a Ulm.

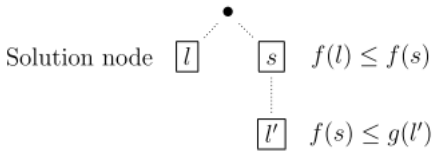


En la parte superior de la figura vemos que los sucesores de Mannheim se generan antes que los sucesores de Wurzburg. La solución óptima Frankfurt-Wurzburg-Ulm se genera poco después en el octavo paso, pero aún no se reconoce como tal. Por lo tanto, el algoritmo aún no termina porque el nodo Karlsruhe (3) tiene un valor f mejor (menor) y por lo tanto, está por delante del nodo Ulm (8) en la línea. Solo cuando todos los valores de f son mayores o iguales al del nodo solución Ulm (8) nos hemos asegurado de tener una solución óptima. De lo contrario, podría haber otra solución con costos más bajos.

Mostremos que esto es cierto en general.

Teorema El algoritmo A^* es óptimo. Es decir, siempre encuentra la solución con el menor costo total si la heurística h es admisible.

En el algoritmo `HEURISTICSEARCH`, cada nodo s recién generado es ordenado por la función `SortIn` de acuerdo con la calificación heurística $f(s)$. El nodo con el valor de calificación más pequeño se encuentra al principio de la lista. Si el nodo l al principio de la lista es un nodo de solución, ningún otro nodo tiene una mejor calificación heurística. Para todos los demás nodos s, entonces es cierto que $f(l) \leq f(s)$. Dado que la heurística es admisible, no se puede encontrar una solución mejor l', incluso después de la expansión de todos los demás nodos como se indica la siguiente figura:



Escrito formalmente implica que:

$$g(l) = g(l) + h(l) = f(l) \leq f(s) = g(s) + h(s) \leq g(l^{'}).$$

La primera igualdad se cumple porque l es un nodo solución con $h(l) = 0$. La segunda es la definición de f. La tercera desigualdad se mantiene porque la lista de nodos abiertos se ordena en orden ascendente. La cuarta igualdad es nuevamente la definición de f. Finalmente, la última desigualdad es la admisibilidad de la heurística, que nunca sobreestima el costo desde el nodo s hasta una solución arbitraria. Por tanto, se ha demostrado que $g(l) \leq g(l^{'})$, es decir, que la solución descubierta l es óptima.

1.3 Planificación de ruta con el algoritmo de búsqueda A^*

Muchos sistemas de navegación para automóviles actuales utilizan el algoritmo A^*. La heurística más simple, pero muy buena, para calcular A^* es la distancia en línea recta desde el nodo actual hasta el destino. El uso de 5 a 60 de los llamados puntos de referencia es algo mejor. Para estos puntos elegidos al azar, las rutas más cortas hacia y desde todos los nodos del mapa se calculan en un paso previo al cálculo. Sea l un punto de referencia, s el nodo actual y z el nodo de destino. También sea $c^*(x, y)$ el costo del camino más corto de x a y. Luego obtenemos para el camino más corto de s a l la desigualdad del triángulo:

$$c^*(s, l) \leq c^*(s, z) + c^*(z, l)$$

Resolver para $c^*(s, z)$ da como resultado la heurística admisible:

$$h(s) = c^*(s, l) - c^*(z, l) \leq c^*(s, z).$$

Se demuestra que esta heurística de puntos de referencia es mejor que la distancia en línea recta para la planificación de rutas. Por un lado, se puede calcular más rápido que la distancia en línea recta. Debido al cálculo previo, las distancias a los puntos de referencia se pueden recuperar rápidamente de una matriz, mientras que las distancias euclidianas deben calcularse individualmente. Resulta que esta heurística reduce aún más el espacio de búsqueda, como se puede ver en la imagen dada en el libro de Ertel, de la izquierda que ilustra el árbol de búsqueda de A^* para planificar una ruta desde Ravensburg a Biberach (dos ciudades en el sur de Alemania). L

Los bordes sin heurística (es decir, con $h(s) = 0$) se trazan en color rojo, las líneas de color verde oscuro muestran el árbol de búsqueda utilizando la heurística de distancia en línea recta y los bordes de la heurística de puntos de referencia con 20 puntos de referencia se trazan en azul.

La imagen de la derecha muestra la misma ruta utilizando una búsqueda bidireccional, donde una ruta de Ravensburg a Biberach y otra en la dirección opuesta se planifican efectivamente en paralelo. Si las rutas se encuentran, dadas ciertas condiciones de la heurística, entonces tenemos una ruta óptima.

En la tabla siguiente se puede encontrar un análisis cuantitativo de los tamaños del árbol de búsqueda y el tiempo de cálculo en una PC. Al observar la búsqueda unidireccional, vemos que ambas heurísticas reducen claramente el espacio de búsqueda. Los tiempos de cálculo son realmente interesantes. En el caso de la heurística por puntos de referencia, vemos que el tiempo de cálculo y el tamaño del espacio de búsqueda se reducen en un factor de aproximadamente 12. El costo de calcular la heurística es, por tanto, insignificante. Sin embargo, la distancia en línea recta da como resultado una reducción del espacio de búsqueda de un factor de 6.6, pero solo una mejora de un factor de 2.2 en el tiempo de ejecución debido a la sobrecarga de calcular la distancia euclidiana.

En el caso de la búsqueda bidireccional, a diferencia de la búsqueda unidireccional, vemos una reducción significativa del espacio de búsqueda incluso sin heurística. Por otro lado, el espacio de búsqueda es más grande que el caso unidireccional para ambas heurísticas. Sin embargo, debido a que los nodos están divididos en dos listas ordenadas en la búsqueda bidireccional, las listas se manejan más rápido y los tiempos de cálculo resultantes son aproximadamente los mismos.

Al planificar una ruta, el conductor se preocupa más por el tiempo de conducción que por la distancia recorrida. Por lo tanto, deberíamos ajustar la heurística y reemplazar la distancia en línea recta $d(s,z)$ con el tiempo $t(s,z) = d(s,z)/v_{\max}$. Aquí tenemos que dividir por la velocidad media máxima, que rebaja la heurística, porque hace que los tiempos estimados heurísticamente sean demasiado pequeños. La heurística por puntos de referencia, por el contrario, se basa en rutas óptimas precalculadas y por lo tanto, no se rebaja. Luego, la búsqueda de una ruta optimizada en el tiempo utilizando la heurística de puntos de referencia es significativamente más rápida que con la distancia en línea recta modificada.

	Unidirectional		Bidirectional	
	Tree Size [nodes]	Comp. time [msec.]	Tree Size [nodes]	Comp. time [msec.]
No heuristic	62000	192	41850	122
Straight-line distance	9380	86	12193	84
Landmark heuristic	5260	16	7290	16

El algoritmo de jerarquías de restricciones, funciona incluso mejor que A^* con heurística de puntos de referencia. Se basa en la idea de combinar, en un paso de pre cálculo, varios bordes en los llamados `shortcut`, que luego se utilizan para reducir el espacio de búsqueda. Al observar la búsqueda unidireccional, vemos que ambas heurísticas reducen claramente el espacio de búsqueda.

1.4 Búsqueda IDA^*

Una búsqueda A^* hereda una peculiaridad del BFS. Tiene que guardar muchos nodos en la memoria, lo que puede llevar a un uso muy alto de la memoria. Además, se debe ordenar la lista de nodos abiertos. Por lo tanto, la inserción de nodos en la lista y la eliminación de nodos de la lista ya no pueden ejecutarse en tiempo constante, lo que aumenta ligeramente la complejidad del algoritmo. Basándonos en el algoritmo heapsort, podemos estructurar la lista de nodos como un heap con complejidad de tiempo logarítmico para la inserción y eliminación de nodos.

Ambos problemas se pueden resolver, de manera similar al BFS, mediante una profundización iterativa.

Trabajamos con el DFS y elevamos el límite sucesivamente. Sin embargo, en lugar de trabajar con un límite de profundidad, aquí usamos un límite para la evaluación heurística $f(s)$. Este proceso se denomina algoritmo IDA^*.

1.5 Comparación empírica de los algoritmos de búsqueda

En A^* o (alternativamente) IDA^*, tenemos un algoritmo de búsqueda con muchas buenas propiedades. Es completo y óptimo. Sin embargo, lo más importante es que funciona con heurística y por lo tanto, puede reducir significativamente el tiempo de cálculo necesario para encontrar una solución. Nos gustaría explorar esto empíricamente en el ejemplo de 8-puzzle.

Para el 8-puzzle hay dos heurísticas simples admisibles. La heurística h_1 simplemente cuenta el número de cuadrados que no están en el lugar correcto. Claramente esta heurística es admisible. La heurística h_2 mide la distancia de Manhattan. Para cada cuadrado, se suman las distancias horizontal y vertical a la ubicación de ese cuadrado en el estado objetivo. Luego, este valor se suma en todos los cuadrados. Por ejemplo, la distancia de Manhattan de dos estados es,

es calculada como:

$$h_2(s) = 1+1+1+1+2+0+3+1 = 10$$

La admisibilidad de la distancia de Manhattan también es obvia (¿cómo?).

Para una comparación con la búsqueda no uniformizada, se aplica el algoritmo A^* con las dos heurísticas h_1 y h_2 y la profundización iterativa a 132 problemas de 8-puzzle generados aleatoriamente. Los valores promedio para el número de pasos y el tiempo de cálculo se dan en la tabla siguiente:

Depth	Iterative deepening		A* algorithm				Num. runs
	Steps	Time [sec]	Heuristic h_1		Heuristic h_2		
			Steps	Time [sec]	Steps	Time [sec]	
2	20	0.003	3.0	0.0010	3.0	0.0010	10
4	81	0.013	5.2	0.0015	5.0	0.0022	24
6	806	0.13	10.2	0.0034	8.3	0.0039	19
8	6455	1.0	17.3	0.0060	12.2	0.0063	14
10	50512	7.9	48.1	0.018	22.1	0.011	15
12	486751	75.7	162.2	0.074	56.0	0.031	12
			IDA*				
14	–	–	10079.2	2.6	855.6	0.25	16
16	–	–	69386.6	19.0	3806.5	1.3	13
18	–	–	708780.0	161.6	53941.5	14.1	4

Vemos que la heurística reduce significativamente el costo de búsqueda en comparación con la búsqueda no uniformizada. Si comparamos la profundización iterativa con A^* con h_1 a la profundidad 12, por ejemplo, se hace evidente que h_1 reduce el número de pasos en un factor de aproximadamente 3000, pero el tiempo de cálculo solo en un factor de 1023. Esto se debe al mayor costo por paso para el cálculo de la heurística.

Un examen más detenido revela un salto en el número de pasos entre la profundidad 12 y la profundidad 14 en la columna de h_1. Este salto no puede explicarse únicamente por el trabajo repetido realizado por IDA^*. Se produce porque la implementación del algoritmo A^* elimina los duplicados de nodos idénticos y por lo tanto, reduce el espacio de búsqueda. Esto no es posible con IDA^* porque casi no guarda ningún nodo. A pesar de esto, A^* ya no puede competir con IDA^* más allá de la profundidad 14 porque el costo de clasificar en nuevos nodos aumenta demasiado el tiempo por paso.

Un cálculo del factor de ramificación efectiva arroja valores de alrededor de 2.8 para una búsqueda no uniformizada. La heurística h_1 reduce el factor de ramificación a valores de aproximadamente 1.5 y h_2 a aproximadamente 1.3. Podemos ver en la tabla que una pequeña reducción del factor de ramificación de 1.5 a 1.3 nos da una gran ventaja en el tiempo de cálculo.

La búsqueda heurística tiene, por tanto, una importancia práctica importante porque puede resolver problemas que están lejos del alcance de una búsqueda no uniformizada.