Semana 14

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

11-2021

Meet in the midde

Meet in the middle

- La técnica de "Encontrarse en el medio" (Meet in the middle) divide el espacio de búsqueda en dos partes de aproximadamente el mismo tamaño, realiza una búsqueda para ambas partes y, finalmente, combina los resultados de las búsquedas.
- Esta técnica nos permite acelerar los resultados de las búsquedas de tiempo $O(2^n)$ para que funcionen en un tiempo $O(2^{\frac{1}{2}})$
- Usando un algoritmo $O(2^n)$, el tamaño máximo que podríamos procesar (por el tiempo de ejecución) es de $n = 20$, mientras que un algoritmo $O(2^{\frac{1}{2}})$ nos permite procesar entradas cercanas a $n = 40$.

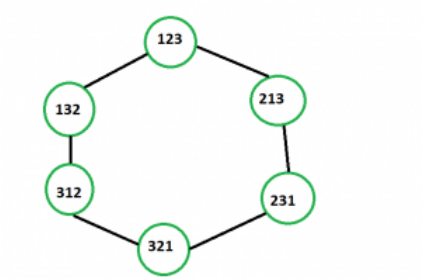
Shortest Path using Meet In The Middle

Data una permutación $P = \{p_1, p_2, \cdots p_n\}$ de los primeros ne números naturales $(1 \leq n \leq 10)$, encontrar el mínimo número de intercambiios para pasar de la permutación P a $P = \{p'_1, p'_2, \cdots p'_n\}$. Solo se pueden intercambiar dos elementos consecutivos, p_i y p_{i+1} $(1 \leq i \leq 10)$.

Este problema se puede resolver utilizando el algoritmo Dijkstra. Parece que no hay nada relacionado con un grafo en la declaración. Pero supongamos que una permutación es un vértice, entonces cada intercambio de los elementos de una permutación es un borde que conecta este vértice con otro vértice. Por lo tanto, encontrar el número mínimo de intercambios ahora se convierte en un simple problema de BFS/ruta más corta.

Input: P = "213", P' = "231"

Output: 2



Ahora analicemos la complejida de tiempo. Tenemos $n!$ vértices, cada vértices tiene $n-1$ vértices adyancentes. Tomando en cuenta esto, la complejidad que tendríamos al aplicar el algoritmo sería $O(n \log n * n!)$. Usando al técnica de "Encontrarse en la mitad" puede hacer que la solución sea más rápida.

Haremos P y P' sean ambos el inicio de nuestra búsqueda. Realizaremos el BFS desde estos puntos, comenzamos y terminamos al mismo tiempo usando solo una fila.

1. Sea *start* el vértice inicial y *finish* el vértice final.
2. $visited[start] = true$ y $visited[finish] = true$.
3. $src[start] = start$ y $src[end] = end$
4. $dist[start] =$ y $dist[start] = 0$
5. Colocamos *start* y *finish* en la fila, marcándolos como visitados.
6. Mientras la fila no está vacía. sacamos el primer elemento. u. Colocamos en al fila todos los elementos adyacentes, v, de u que hayan visitados. Si algún v ya fue visitado, si $src[u] \neq srv[v]$ entonces regreamos $dist[u] + dist[v] + 1$.

Ver código bfs_middle.cpp

El problema de la suma de subconjuntos

Retomemos el problema de la suma de subconjuntos:

- Dado un conjunto S de N números enteros positivos, encontrar los subconjuntos que sumen la cantidad C. Si no se encuentra algún subconjunto que cumple, se debe responder con el conjunto vacío.

La idea es dividir nuestro conjunto en dos conjuntos A y B, de modo que ambos conjuntos contengan aproximadamente la mitad de los números. Realizamos dos búsquedas: la primera búsqueda genera todos los subconjuntos de A y almacena sus sumas en una lista S_A, y la segunda búsqueda crea una lista S_B similar para B. A continuación, basta con verificar si podemos elegir un elemento de S_A y otro de S_B, tal que la suma sea C.

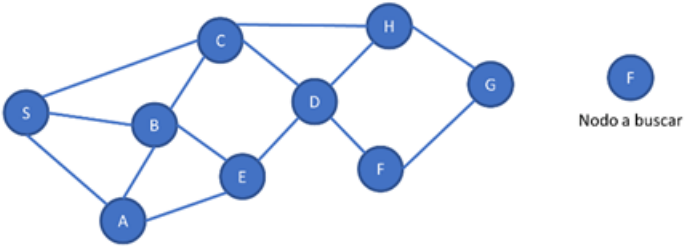
Por ejemplo, veamos cómo se procesa el conjunto $S = \{2, 3, 7, 9\}$. Primero, dividimos el conjunto en conjuntos $A = \{2, 3\}$ y $B = \{7, 9\}$. En seguida, creamos las listas $S_A = [\{\} = 0, \{2\} = 2, \{3\} = 3, \{2, 3\} = 5]$ y $S_B = [\{\} = 0, \{7\} = 7, \{9\} = 9, \{7, 9\} = 16]$. Dada que S_A contiene la suma 2 y S_B contiene la suma 7, concluimos que el conjunto original tiene un subconjunto con suma $2 + 7 = 9$.

Con una buena implementación, podemos crear las listas S_A y S_B en tiempo $O(2\frac{n}{2})$, de la tal forma que las listas estén ordenadas. Después de, podemos usar la técnica de dos punteros para verificar en tiempo $O(2\frac{n}{2})$ si la suma C puede ser creada a partir de S_A y S_B. Por tanto, la complejidad del algoritmo es $O(2\frac{n}{2})$.

Búsqueda A*

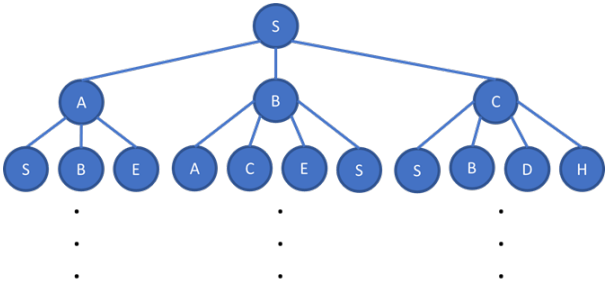
Búsqueda A*

Una de las principales operaciones que se pueden hacer sobre un grafo es buscar un nodo específico, es decir, aquel que contiene la información que se desea encontrar.



- La forma ingenua de realizar esta búsqueda es moverse de un nodo a otro hasta que se logre encontrar el nodo deseado, o se visiten todos y no se encuentre el nodo en cuestión. En este último caso, el algoritmo contestará que el nodo no existe en el grafo. Esta búsqueda ingenua parece muy simple, pero no lo es, debido a que el grafo es una estructura de datos no lineal y no tiene un inicio o un fin.
- Además de que se tiene que marcar cuál nodo ya visitamos para no regresar a él y caer un ciclo.

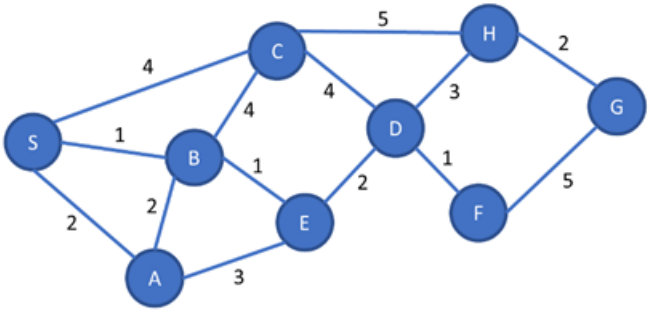
- Se puede hacer que la búsqueda parta de un nodo y, en caso de no ser el buscado, se proceda a buscar en los vecinos de este nodo, y luego, en los vecinos de los vecinos, y así sucesivamente. Lo anterior, garantizará recorrer todos en algún momento, siempre y cuando el grafo sea finito.
- Esta búsqueda genera una estructura en forma de árbol, al que se le conoce como **árbol de búsqueda**, en el que su raíz es el nodo donde se inicia la búsqueda y su factor de ramificación depende del número de vecinos que tengan los nodos.



El nodo que se está visitando en un momento dado de la búsqueda se conoce como nodo actual. Si el nodo actual no es la solución, se procede a visitar a sus vecinos, quitando los que ya fueron visitados. A esa acción se le conoce como expandir el nodo, es decir, generar todos sus vecinos y tenerlos listos para seguir la búsqueda. Un nodo visitado ya fue expandido. Todos los algoritmos de búsqueda sistemática realizan esa acción, la única diferencia entre ellos es la decisión de cuál de los vecinos se debe visitar (o explorar) a continuación.

Hasta el momento conocemos tres algoritmos de búsqueda sistémica:

- **BFS, DFS**. EL BFS usa una fila, lo que garantiza que todos los del mismo nivel se exploran primero. El DFS usa una pila, lo que garantiza que el más profundo se explora primero. Ambos algoritmos realizan búsquedas "ciegas".



- Para ellos, existe un algoritmo sistemático que intenta aprovechar la información de las aristas. Este algoritmo es una modificación de BFS, el cual usa una función $c(n)$, donde n es el nodo, llamada costo, que calcula el costo que hay de la raíz al nodo actual, sumando los valores de las aristas por las que se ha pasado, los cuales son considerados costos.

En general, a este tipo de búsquedas se les conoce como **Búsqueda de costo uniforme** (UCS, por sus siglas en inglés, Uniform Cost Search). En estas búsquedas, se expande primero el nodo de menor costo (BFS expande primero el de menor profundidad), el cual es calculado con la función $c(n)$, donde n es el nodo.

Posteriormente, se pensó que se podrían generar algunos algoritmos de búsqueda sistemática que fueran más eficientes al considerar alguna información extra sobre el problema que se tuviera a la mano y que diera alguna idea de cuál es el nodo más prometedor para ser explorado. Así aparecieron los algoritmos heurísticos, los cuales utilizan una función para estimar la distancia a la que un nodo específico está del objetivo. Esta función es una función heurística $h(n)$, donde n es el nodo, lo que les da su nombre.

- **Búsqueda Primero el Mejor** (BFS, por sus siglas en inglés, Best First Search), expande primero el que se estima que le falta menos para llegar al objetivo, esto es, el que tenga un menor valor de la función heurística $h(n)$.

- El problema principal de los algoritmos heurísticos es, precisamente, el diseño de la función heurística $h(n)$. No es simple diseñar una buena función heurística, ya que, normalmente esta depende del tipo de problema que se está resolviendo.
- Si analizamos lo que hemos visto, podemos observar que UCS no toma en cuenta lo que falta para llegar al objetivo y BFS no toma en cuanta lo que se lleva recorrido.
- ¿Sería posible combinar ambas estrategias? Los teóricos de los antiguos astronautas piensan que sí.

Existe un algoritmo llamado A* que utiliza una combinación entre UCS y BFS, para estimar el camino completo, usando una nueva función $f(n) = c(n) + h(n)$, donde $c(n)$ es el costo de UCS y $h(n)$ es la heurística de BFS. A* expande primero los nodos que tenga una menor $f(n)$, es decir, el nodo con el camino completo desde la raíz al objetivo, con el menor costo estimado.

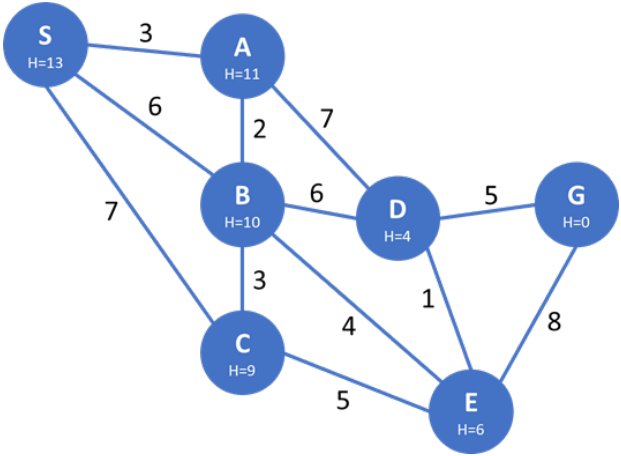
A* tiene una propiedad interesante. Si $h(n)$ cumple ciertas características, A* es óptimo, esto es, logra encontrar el camino más corto para ir del nodo inicial al nodo objetivo. Las características que debe cumplir la heurística $h(n)$ son:

- **Admisible:** Es una heurística optimista, es decir, aquella que regresa un costo estimado menor que el que realmente es.
- **Consistente:** Si para cada nodo n y para cada nodo vecino v, el costo estimado para alcanzar la meta desde n es menor que el costo del paso para ir de n a v, más el estimado para ir de v a la meta, es decir, si $h(n) < g(n, v) + h(v)$, donde $g(n, v)$ es el costo real para ir del nodo n al nodo v (peso de la arista).

- Una forma de garantizar que la heurística es admisible es usar como función heurística la solución para el problema relajado, es decir, para el problema original al que se le quitan una o más restricciones.
- La complejidad de A* depende completamente de la heurística que se utilice, por lo que no tienen sentido hablar de la complejidad de A* como algoritmo.

Una aplicación de A*

- Si usamos A* para resolver el problema de encontrar la distancia más corta de un nodo a otro en un grafo que representa ciudades, una heurística que cumple con ser admisible y consistente es la distancia en línea recta de una ciudad a otra. La distancia recta, siempre es menor que la distancia real y cumple con la desigualdad necesaria para ser consistente.


S
H=13
A
H=11
B
H=10
D
H=4
G
H=0
C
H=9
E
H=6
3
6
7
2
7
6
5
3
1
4
8
5

Búsqueda IDA* (Iterative Deepening A*)

Búsqueda IDA* (Iterative Deepening A*)

El algoritmo A* tiene el mismo inconveniente de DFS, esto es, se puede ir a explorar algo sumamente profundo cuando en la rama de al lado podría encontrar la solución más rápidamente, esto es, a una profundidad menor. Este problema en DFS se puede minimizar usando un algoritmo con profundidad iterativa, esto es, se pone un límite L a la profundidad, el cual inicia en 1, para la exploración y si en ese límite no se encuentra la solución, se aumenta L en 1 y se vuelve a empezar. Aunque esto es muy tardado, permite encontrar con DFS las soluciones menos profundas. A esta variante de DFS se le conoce como DFS con profundidad iterativa.

Procedure 1 IDA*

Input: $v : Vertex, g : Graph$

$L = 1$

while L has not reached the end **do**

$result \leftarrow A*(v, g, L)$

if $A*$ gets results **then**

return *result*

end if

end while

return *NULL*