PRÁCTICA 4 - Backtracking y Branch-and-Bound

Un repartidor muy eficaz

¿EN QUÉ CONSISTE BACKTRACKING? ¿FUERZA BRUTA?

Existen problemas para los que no existe una forma que nos permita obtener una solución de manera precisa y rápida. La manera de resolverlos consiste en realizar una búsqueda exhaustiva entre todas las posibles soluciones hasta dar con una o un grupo de soluciones válido. La búsqueda exhaustiva en un espacio limitado dado se conoce como fuerza bruta y consiste en ir probando todas las soluciones potenciales una por una hasta encontrar una solución satisfactoria, o bien agotar las posibilidades. El problema viene en que normalmente la fuerza bruta es inviable para espacios de soluciones grandes lo cual ocurre bastante a menudo.

Sin embargo, uno de los algoritmos que vamos a utilizar en esta práctica es del de Backtracking "vuelta atrás". Es una mejora del algoritmo de fuerza bruta debido a que la búsqueda se realiza de manera estructurada, descartando grandes bloques de soluciones para reducir el espacio de búsqueda.

PRINCIPAL DIFERENCIA ENTRE FUERZA BRUTA Y BACKTRACKING

La diferencia principal entre ambos algoritmos es que en el primero la estrategia consiste en probar una solución potencial tras otra sin ningún criterio. Es decir, se irán comprobando una a una lo que llevará a que se tarde en alcanzar la solución o que directamente no se alcance.

No obstante, en el segundo algoritmo las soluciones se forman de manera progresiva, generando soluciones parciales y comprobando en cada paso si la solución que se está obteniendo puede conducir a una solución satisfactoria. Básicamente, con este algoritmo eliminamos la necesidad de llegar hasta todas las hojas del árbol y como consecuencia se obtenga antes la solución.

¿EN QUÉ CONSISTE BRANCH-AND-BOUND?

Una de las características de Branch & Bound en comparación con el backtracking es la forma en que se explora el árbol. El backtracking generalmente tiene como objetivo profundizar, buscando aquellas ramas que conducen a una solución, pero siempre prioriza la profundización sobre la expansión. El algoritmo Branch & Bound propone otra forma de explorar el árbol, el ancho. Aunque es posible pensar que eventualmente conducirá al mismo resultado, no es así. La principal tarea que persiguen los algoritmos de ramificación y poda es priorizar las ramas candidatas con mayor probabilidad de éxito, es decir, intentar ramificar aquellas ramas con mayor probabilidad de éxito estimado.

Suelen ser usados principalmente para problemas de optimización (maximizar o minimizar una función) y aunque cabe esperar mejores resultados que el algoritmo de backtracking, no siempre es así, porque depende mucho de los procedimientos que se desarrollen para optar por las ramas más adecuadas. De hecho, pueden incluso ser peores que los algoritmos de backtracking, algo que no debe sorprendernos. Sin embargo, por norma general, suelen ser algo mejores que estos ya que persiguen buscar las cotas de poda, permitiendo podar el árbol lo antes posible y evitar búsquedas innecesarias.

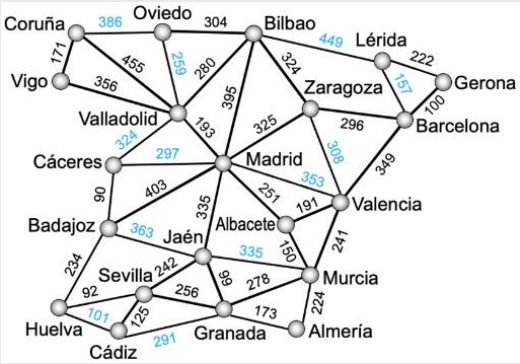
DIFERENCIAS ENTRE BACKTRACKING Y BRANCH-AND-BOUND

En backtracking, en cuanto se genera un nuevo hijo del nodo en curso, este hijo pasa a ser el nodo en curso. Por otro lado, en el segundo algoritmo, se generan todos los hijos del nodo en curso antes de que cualquier otro nodo pase a ser el nuevo nodo en curso (no se va a realizar un recorrido en profundidad).

La consecuencia de esto es que, en el primer algoritmo, los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso. En el segundo algoritmo, puede haber más nodos vivos, que se almacenan en una lista de nodos vivos.

PLANTEAMIENTO BACKTRACKING

Lo primero que nos planteamos es qué estructuras íbamos a utilizar para llevar a cabo el problema. Inicialmente solo tenemos un simple grafo (HashMap o TreeMap vistos en EDAI).



Tras informarnos, nos decidimos por usar tres estructuras que nos ayudarían a resolver el problema:

Visitados: Un mapa con clave el vértice y como valor un booleano que nos indica si el vértice ha sido o no visitado.

Resultados: Pensamos en usar un ArrayList<ArrayList<String>> o directamente un ArrayList<Camino>, siendo Camino una clase que contiene la lista de vértices que compondrán un camino completo.

Etapa: Un array de tipo String que va a ir almacenando cada uno de los vértices en cada etapa de profundidad del algoritmo en el arbol de exploración.

Además, recurrimos al uso de una variable 'Nivel' que indicará el nivel de profundidad en el que se encuentra nuestro algoritmo en cada etapa.

Ya disponemos de las estructuras necesarias para pasar a la segunda fase. Sin embargo, a estas estructuras es necesario establecerle unos valores iniciales.

El mapa de visitados lo vamos a rellenar con todos los vértices que van a conformar el grafo y le asociamos un valor por defecto. El valor asociado va a ser un tipo booleano, inicialmente a false. Esta estructura nos va a servir de guía para poder observar qué vértices son los que hemos visitado (y por tanto no podemos volver a escoger) y cuáles están sin visitar (siendo estos los posibles candidatos a explorar en cada etapa en nuestro árbol).

Para rellenarlo, recorremos el conjunto de vertices de nuestro grafo y los iremos insertando en este mapa junto con un valor asociado a false:

A	false
B	false
C	false
D	false
E	false
F	false

Ahora nos planteamos la opción de que nuestro punto de partida pueda ser considerado como visitado. Supongamos que partimos desde el vértice B, por lo que tendremos que establecer en ese mapa un valor de true.

A	false
B	true
C	false
D	false
E	false
F	false

Además, vimos que otro de los pasos que debíamos hacer era establecer en nuestra segunda estructura auxliar. Esta estructura inicialmente tendría valores a null para todas las posiciones, por lo que solo deberemos sustituir la posición 0 y la última con nuestro vértice de partida.

null	null	null	null	null	null	null

Esta estructura tiene un total de n+1 posiciones, siendo N el número total de vértices del grafo. Ahora, nos disponemos a establecer la primera y última posición con el valor del vértice de partida, para nuestro caso visual B.

B	null	null	null	null	null	B

La estructura Resultados va a permanecer vacía, puesto que hasta que no se llame al algoritmo de Backtracking y se encuentren soluciones, no se agregarán.

Ahora que las estructuras están inicializadas, llamaremos a nuestro algoritmo Backtracking recursivo.

Tendremos un método público **public List BackTracking()** y un método privado.

Para el privado, recibirá el array Etapa creado e inicializado anteriormente, el nivel de profundidad en el que nos encontramos explorando, el mapa de vertices visitados y la estructura de resultados donde se guardarán las soluciones.

En cada llamada recursiva se contemplará como candidatos todos los nodos de nuestro grafo, por lo que podemos recorrer todos los nodos de nuestro mapa de visitados. Sin embargo, sólo consideraremos como nodo a valorar aquel que no esté visitado.

Una vez elegido nuestro nodo candidato, es necesario valorar si es viable agregarlo a la solución, es decir, si podemos aceptarlo. Consideraremos que un nodo se puede aceptar si desde el último vértice que agregamos a la solución parcial podemos alcanzar este nuevo vértice candidato.

Sólo en el caso de que se haya aceptado a este nuevo candidato, se agregará a la solución parcial Etapa y sólo quedará comprobar si se ha llegado a una solución completa o si, por el contrario, es necesario seguir profundizando en el árbol para intentar llegar a ella.

Cuando hayamos llegado a un nivel de profundidad igual al número de vértices menos 1, es decir, cuando Nivel == n-1. En tal caso, agregaremos la solución parcial (Etapa) a la estructura Resultado. De esta manera sabemos si hemos llegado a una solución completa.

Si se da el caso de que la solución no sea completa, sabemos que tendremos que profundizar…

Para ello, marcaremos el vértice candidato aceptado como visitado en nuestro mapa de visitados, haremos una llamada recursiva al algoritmo, teniendo en cuenta que en esta ocasión le pasamos el n+1. Cuando finalice la llamada recursiva, desmarcaremos como visitado el nodo candidato aceptado en el primer paso.

Sólo en el caso de que el candidato sea el último nodo antes de la vuelta al punto de partida, se deberá comprobar si permite también cerrar el ciclo, verificando si existe una arista desde el nodo candidato al nodo origen.

Con esto resumiríamos la resolución para nuestro algoritmo backtracking recursivo capaz de obtener todos los posibles caminos que visitan cada vértice una única vez y vuelve al punto de partida.

Para cada llamada recursiva de nuestro algoritmo de backtracking recorremos todos los nodos y comprobamos si el nodo está visitado o no lo está y si es alcanzable desde el último nodo agregado a la solución parcial.

Esto va a suponer que, si nuestro arbol tiene n nodos, en cada llamada recursiva estaremos haciendo un bucle de n iteraciones para comprobar si podemos escogerlo.

PLANTEAMIENTO BRANCH-AND-BOUND

Vamos a usar una clase para ejecutar propiamente el algoritmo y una segunda clase que actúa como estructura secundaria para crear objetos que representen un camino. La segunda clase será nuestra base para poder implementar el algoritmo principal. Esta clase consta de una lista donde se almacenarán de forma consecutiva los vértices que formarán el camino, un valor entero que indicará el número de vértices que se han visitado y el coste total de las aristas que forman el camino actual y un valor de coste estimado para el camino.

Ahora nos quedaría definir el algoritmo propio de Branch-AND-Bound. Desarrollaremos dentro un método que no permitirá obtener el valor de arista mínimo de todo el grafo.

También definiremos una estructura auxiliar donde se irán almacenando los caminos parciales. Posteriormente definiremos también la estructura que albergará el resultado final.

ESTUDIO EXPERIMENTAL

A la hora de ejecutar los los dos algoritmos y compararlos para mapas con mayor y menor número de vértices y aristas, hemos llegado a la conclusión de que BackTracking es una opción mejor que Branch&Bound cuanto mayor es el mapa a explorar.

Pese a que BackTracking implemente una búsqueda en profundidad inevitable mientras que Branch&Bound puede prescindir de esta, una vez realizada resuelve recursivamente los problemas restantes en función de la primera solución obtenida.

NumVertices	NumAristas	TB&B(ms)	TBackTracking(ms)
1000	1998	264	63
1000	101698	123	51
1000	201398	96	32
1000	301098	127	47
1000	400798	96	32
1000	500498	111	28
1000	600198	96	32
1000	699898	114	29
1000	799598	126	43
1000	899298	128	44
1000	998998	134	42
1000	999000	102	32

Bibliografía

https://www.cartagena99.com/recursos/alumnos/apuntes/Capitulo6.pdf

https://elvex.ugr.es/decsai/algorithms/slides/5%20Backtracking.pdf

https://www.javatpoint.com/branch-and-bound-vs-backtracking