PRÁCTICA 4- BRACKTRACKING Y BRANCH-AND-BOUND

Augustin Alexandru Besu
Javier Lacal Pelegrin

ÍNDICE

INTRODUCCIÓN

BACKTRACKING

En ciertas circunstancias, hay problemas para los que no se conoce un algoritmo para su resolución o al menos, o no contamos con un algoritmo eficiente para calcular su solución. En estos casos, la única posibilidad es una exploración directa de todas las posibilidades. La técnica Backtracking es un método de búsqueda de soluciones exhaustiva sobre grafos dirigidos acíclicos, el cual se acelera mediante poda de ramas poco prometedoras.

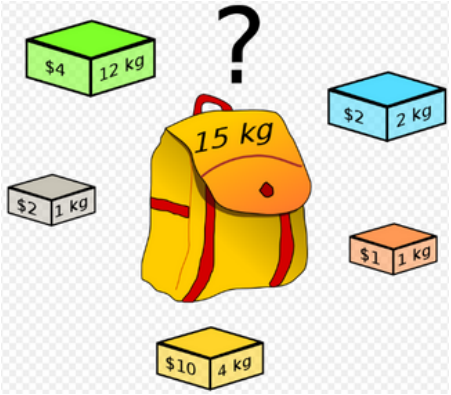
Es decir, otra manera de verlo podría ser diciendo que:

> Se representan todas las posibilidades en un árbol.
> Se resuelve buscando la solución por el árbol (de una determinada manera).
> Hay zonas que se evitan por no contener soluciones (poda).
> La solución del problema se representa en una lista ordenada (X1, X2,..., Xn) (no llenando necesariamente todas las componentes).
> Cada Xi se escoge de un conjunto de candidatos.
> A cada lista se le llama estado.
> Se trata de buscar estados solución del problema.

Como todo problema que hemos ido abordando, tiene dos posibles estados de parada:

a) cuando se alcanza un estado solución.
b) cuando se alcanzan todos los estados solución.

Ejemplos claros para dicha técnica puede ser el que ya vimos en su día con el problema de la mochila, donde mostraríamos en una lista ordenada los objetos que Pedro podría llevar en su mochila para maximizar el beneficio de los mismos, acabando dicho problema con la solución, de haber llegado al beneficio máximo de objetos y/o no le queden más objetos por explorar. Otro caso que puede estar más presente en nuestras vidas es el juego del ajedrez, donde se busca en los mínimos movimientos posibles llegar a la solución, que sería el famoso 'Jaque-Mate'.



INTRODUCCIÓN

BRANCH-AND-BOUND

Es una generalización de la técnica de de la técnica de backtracking. Para ello, al igual que hacíamos en dicha técnica, se realiza un recorrido sistemático del árbol de estados de un problema, si bien ese recorrido no tiene por qué ser en profundidad: usando una estrategia de ramificación.

Además, utilizaremos técnicas de poda para eliminar todos aquellos nodos que no lleven a soluciones óptimas (estimando, en cada nodo, cotas del beneficio que podemos obtener a partir del mismo). Antes de pasar a comentar dicha técnica con una ejemplo, vamos a explicar las mayores diferencias respecto con BT; en bactracking, tan pronto como se genera un nuevo hijo, este hijo pasa a ser el nodo en curso. Sin embargo en B&B, se generan todos los hijos del nodo en curso antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en curso (no se realiza un recorrido en profundidad).

EJEMPLO:
Para poder aclararnos mejor, vamos a explicar dicho ejemplo con un juego (el 8-puzzle), el cual podríamos resolver con dicha técnica. Dicho juego consiste en que, dado un tablero de 3×3 con 8 fichas (cada ficha tiene un número del 1 al 8) y un espacio vacío. El objetivo es colocar los números en los mosaicos para que coincidan con la configuración final utilizando el espacio vacío. Podemos deslizar cuatro fichas adyacentes (izquierda, derecha, arriba y abajo) en el espacio vacío. Vamos a imaginarnos que empezamos con la siguiente configuración inicial y queremos que el juego acabe con esta otra:

CONFIGURACIÓN INICIAL

1	2	3
5	6	
7	8	4

CONFIGURACIÓN FINAL (SOLUCIÓN)

1	2	3
5	8	6
	7	4

Podemos realizar una búsqueda en amplitud en el árbol de espacio de estado. Esto siempre encuentra un estado objetivo más cercano a la raíz. Pero no importa cuál sea el estado inicial, el algoritmo intenta la misma secuencia de movimientos como DFS (algoritmo de fuerza bruta). Si hacemos esta expansión, tendríamos un árbol con estas posibles opciones

INTRODUCCIÓN

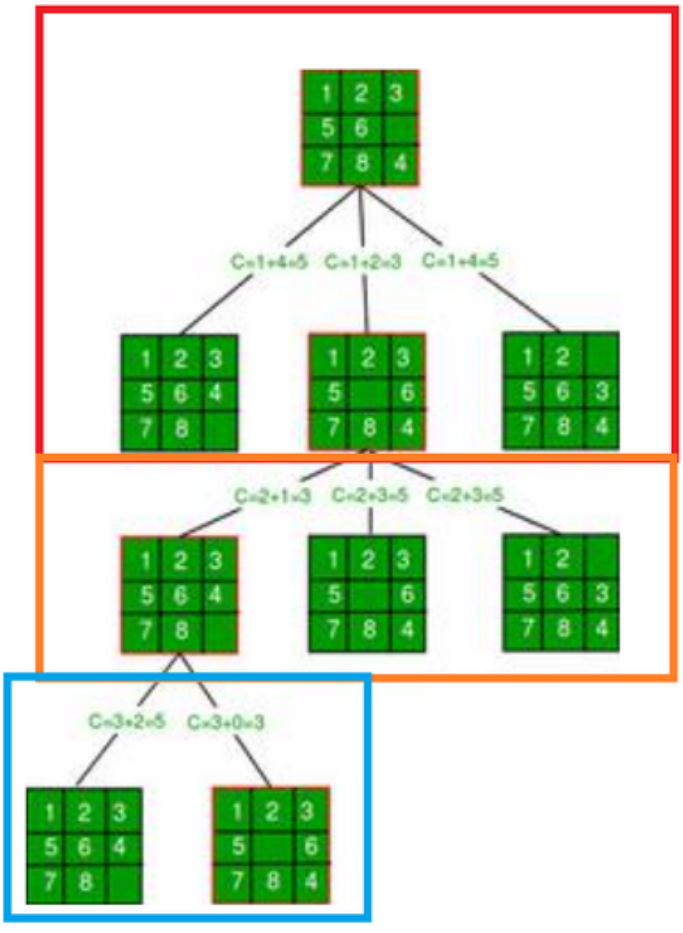
BRANCH-AND-BOUND

Básicamente, hay tres tipos de nodos involucrados en Branch and Bound
1. El nodo en vivo es un nodo que se ha generado pero cuyos hijos aún no se han generado.
2. E-node es un nodo vivo cuyos hijos se están explorando actualmente. En otras palabras, un nodo E es un nodo que se está expandiendo actualmente.
3. El nodo muerto es un nodo generado que no debe expandirse ni explorarse más. Todos los hijos de un nodo muerto ya se han expandido.

Función de coste. Cada nodo X del árbol de búsqueda está asociado a un coste. La función de costo es útil para determinar el próximo nodo E. El siguiente E-nodo es el que tiene el menor costo. La función de costo se define como:

```
C(X) = g(X) + h(X) donde
g(X) = costo de llegar al nodo actual
       desde la raíz
h(X) = costo de llegar a un nodo de respuesta desde X.
```

Ahora bien, desglosamos el problema y lo vamos explicando:



Este sería el camino que iría tomando el nodo, teniendo en cuenta que solo se toma el que tenga menos coste por la función que hemos definido previamente

OBTENCIÓN DE TIEMPOS

TABLAS COMPARATIVAS

Decir que todo el código está accesible y explicado en nuestro repositorio de Git.
Para poder obtener estas tablas comparativas, hemos tomado los tiempos en ejecución en Java (puede verse en la clase "Generador" como hemos obtenido dichos resultados), y los resultados los hemos arrojado en un .txt que está subido a Git con el nombre 'TiemposAlgoritmos.txt'. Si lo reflejamos en tablas los resultados han sido (incluyendo los de generar el archivo:

-Tiempo generación archivos:

Nº vértices	Tiempo Generación (ms)
5	3
10	5
15	6
20	8

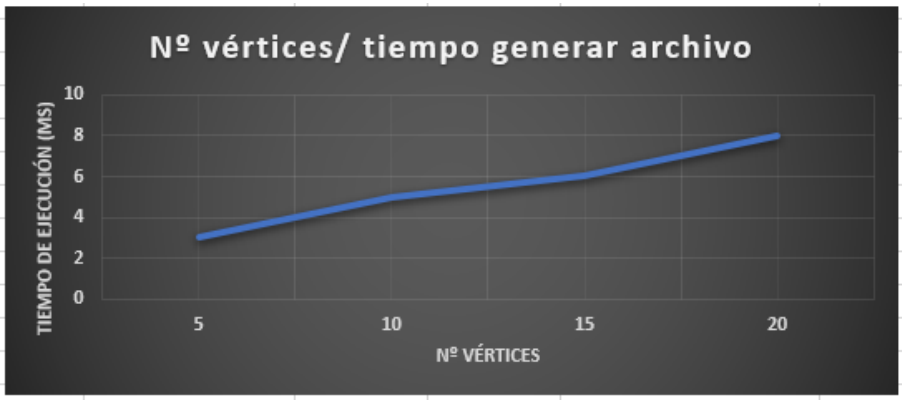
-Tiempos BT y B&B:

Nº vértices	Tiempo BT (ms)	Tiempo B&B (ms)
5	3	1
10	93	17
15	2709	301
20	45056	36115

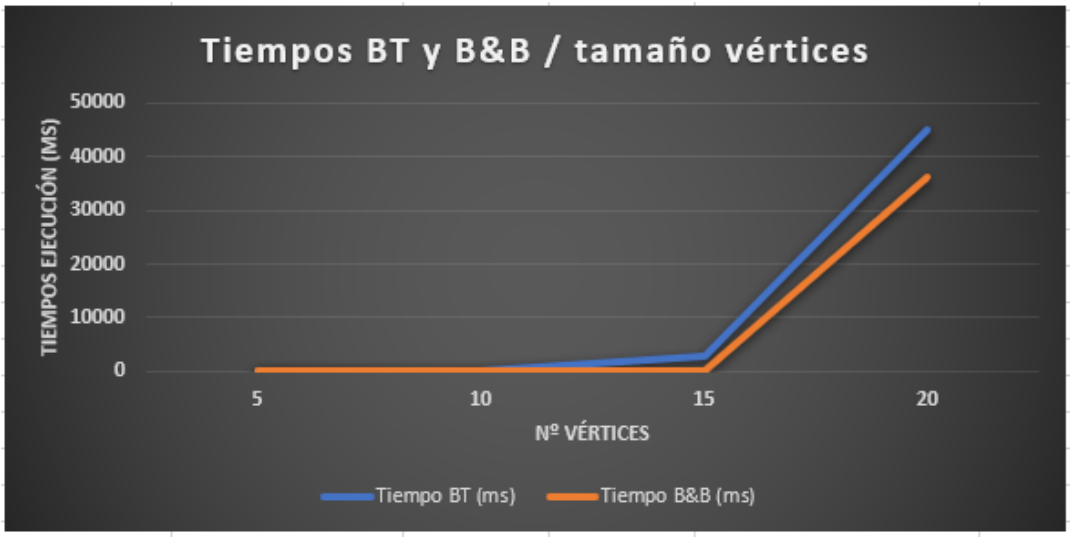
Primeramente, antes de sacar una breves conclusiones sobre dichos tiempos, los obtenidos son subjetivos al ordenador que se haya empleado. En nuestro caso, hemos empleado el portátil más potente. Ahora bien, primeramente debemos de comentar que los archivos con los respectivos vértices se pueden encontrar también en nuestro repositorio de GitHub con el nombre 'Prueba05Vertices.txt', 'Prueba10Vertices.txt', 'Prueba15Vertices.txt' y 'Prueba20Vertices.txt' respectivamente. En lo que a los tiempos de generación se refiere, aunque la diferencia no es muy notable, podemos destacar que a menos número de vértices, mayor es el número de vértices, menos tarda en generarse dicho archivo. Estos tiempos han sito tomados en la clase 'Generador.java'. Por otra parte, y es donde está la parte interesante, es la ejecución con n vértices con los dos algoritmos a tratar. Empezaremos con el caso del BT, el cual es mucho más tardío en comparacion al B&B. Esto imaginamos que puede ser debido a los funcionamientos de dichos algoritmos, y es que Branch and Bound va explorando y si es posible, podando, lo cual hace que sea más rápido. Por último, recalcar que no hemos cogido más tamaño de vértices porque nos daba error de Stack Oveflow.

A partir de los anteriores tiempos obtenemos estas gráficas:

-Generar archivo:



-BT Y B&B:



Como veníamos diciendo, estos resultados se han producido porque podíamos preveer que uno de los dos algoritmos fuera de un tiempo de ejecución mayor al otro. A continuación, en el estudio computacional de ambos algoritmos veremos como efectivamente esto es así.

ESTUDIO COMPUTACIONAL

Para el problema del Backtracking uno de los factores que determinan el tiempo de ejecución es el n° de nodos generados.

En el peor caso, si el n° de nodos es 2^n o n!, el tiempo de ejecución del algoritmo será generalmente de orden O(p(n)2^n) u O(q(n)n!) respectivamente, con p y q polinomios en n.

La técnica Branch&Bound da lugar a algoritmos de complejidad exponencial, por lo que normalmente se utiliza en problemas complejos que no pueden resolverse en tiempo polinómico.

En caso promedio, el Algoritmo de Branch&Bound obtiene mejores resultados que el Algoritmo de BackTracking, principalmente por la amplitud que tiene de nodos vivos, pero generalmente el tiempo del Branch&Bound es considerablemente peor que el del BackTracking

El pseudocódigo para este primera técnica (Branch and Bound), es el siguiente

```
Funcion RyP {
 P = Hijos(x,k)
 while ( no vacio(P) )
    x(k) = extraer(P)
    if esFactible(x,k) y G(x,k) < optimo
        si esSolucion(x)
             Almacenar(x)
        else
             RyP(x,k+1)
```

Vamos a empezar definiendo cada una de las variables empleadas:

- G(x) es la función de estimación del algoritmo.
- P es la pila de posibles soluciones.
- esFactible es la función que considera si la propuesta es válida.
- esSolución es la función que comprueba si se satisface el objetivo.
- óptimo es el valor de la función a optimizar evaluado sobre la mejor solución encontrada hasta el momento.
- Como podemos tener tanto problemas de maximización, como de minimización, de ahí a que usemos el > y < respectivamente.

En definitiva, debemos de prestar especial atención a la pila, ya que mientras no se encuentre vacía, podremos ir extrayendo valores y comprobar si son solución o no, para almacenarlas.

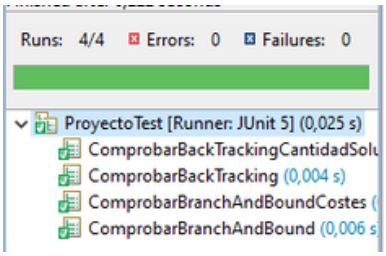
```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
            removeValue(val, n);
```

ocódigo de Backtracking:

Si encuentra una solución, se añade un elemento mas a la solución encontrada y se muestra por pantalla, en el caso de que la solución encontrada sea mayor o igual que el numero de soluciones que habíamos indicado, significara que el algoritmo ha encontrado todas las soluciones y por tanto salimos del método. Vamos iterando todos los elementos, y si se da el caso en el caso de que el valor sea valido para una etapa determinada, aplicamos el valor y hacemos otra llamada recursiva pasando a la siguiente etapa, una vez que encuentra la solución elimina el valor.

TESTS

Para comprobar que todo el trabajo realizado funciona correctamente, vamos a realzar unos test empleando para ello un archivo que también dejaremos en nuestro repositorio de GitHub y que será ¡graphPrimKruskal.txt'. El resultado de todos ellos han sido:



TEST 1

```
@Test
public void ComprobarBackTracking() {

    ProyectoFinal proceso = new ProyectoFinal (ruta + "graphPrimKruskal.txt", "A");

    proceso.BackTracking();

    String esperado = "{[A, C, B, E, F, D, A]=98.0, [A, B, E, C, F, D, A]=110.0, [A, C,

    assertEquals(esperado, proceso.getSolucionFinal().toString());
}
```

TEST 2

```
@Test
public void ComprobarBackTrackingCantidadSoluciones() {

    ProyectoFinal proceso = new ProyectoFinal (ruta + "graphPrimKruskal.txt", "A");

    proceso.BackTracking();

    assertEquals(10, ProyectoFinal.tamArray);
}
```

TEST 3

```
@Test
public void ComprobarBranchAndBound() {

    ProyectoFinal proceso = new ProyectoFinal (ruta + "graphPrimKruskal.txt", "A");

    String esperado = "[A, C, D, F, E, B, A]";

    assertEquals(esperado, proceso.TSPBaB("A").toString());
}
```

TEST 4

```
@Test
public void ComprobarBranchAndBoundCostes() {

    ProyectoFinal proceso = new ProyectoFinal (ruta + "graphPrimKruskal.txt", "A");

    proceso.TSPBaB("A");

    assertEquals("94.0", ProyectoFinal.costes+"");
}
```

CONCLUSIONES Y BIBLIOGRAFÍA

En general, hemos visto a lo largo de esta presentación que ambas técnicas en general son bastante ineficientes, pero en caso promedio, el Algoritmo de Branch&Bound obtiene mejores resultados que el Algoritmo de BackTracking pero generalmente el tiempo del Branch&Bound es considerablemente peor que el del BackTracking

La bibliografía empleada para haber podido realizar este trabajo ha sido: