Práctica 04

Backtracking y Branch-and-Bound.
Un repartidor muy eficaz

Estructura de Datos y Algoritmos II

Rafael Expósito Mullor
Luis Díaz González
Universidad de Almería

INDICE

1. OBJETIVOS

- Construir soluciones a un problema utilizando los métodos algorítmicos backtracking (vueltra atrás) y branch-and-bound (ramificación y poda).

2. INTRODUCCIÓN

2.1. Backtracking

Hay problemas para los que no se conoce un algoritmo para su resolución o al menos, no cuentan 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:

- 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 (X_1, X_2,..., X_n) (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.
- Tiene condiciones de parada:

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

En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. El objetivo del recorrido es encontrar soluciones para algún problema.

El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas).

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.

Los problemas que deben satisfacer un determinado tipo de restricciones son problemas completos, donde el orden de los elementos de la solución no importa. Estos problemas consisten en un conjunto (o lista) de variables a la que a cada una se le debe asignar un valor sujeto a las restricciones del problema. La técnica va creando todas las posibles combinaciones de elementos para obtener una solución. Su principal virtud es que en la mayoría de las implementaciones se puede evitar combinaciones, estableciendo funciones de acotación (o poda) reduciendo el tiempo de ejecución.

La técnica backtracking (vuelta atrás) está muy relacionada con la búsqueda binaria.

2.2. Branch & Bound

Branch & Bound es un método general de búsqueda que se aplica de la siguiente forma:

1. Explora un árbol comenzando a partir de un problema raíz y su región factible (inicialmente, el problema original, con su espacio de soluciones completo).
2. Aplica funciones de acotación al problema raíz, para el que establece cotas inferiores y/o superiores.
3. Si las cotas cumplen las condiciones que se hayan establecido, habremos encontrado la solución óptima del problema y la búsqueda termina.
4. Si se encuentra una solución óptima para un subproblema concreto, ésta será una solución factible para el problema completo, pero no necesariamente su óptimo global.
5. Cuando en un nodo (subproblema), su cota local es peor que el mejor valor conocido en la región, no puede existir un óptimo global en el subespacio de la región factible asociada a ese nodo y, por tanto, ese nodo puede ser eliminado (podado).

En este algoritmo, la búsqueda continua hasta que:

1. Se examinan o “podan” todos los nodos.
2. Se cumple algún criterio preestablecido sobre el mejor valor encontrado y las cotas locales de los subproblemas aún no resueltos.

Este tipo de algoritmos suelen ser de orden exponencial en su peor caso.

2.3 Diferencias entre Backtracking y Branch-and-Bound

En Backtracking, tan pronto como se genera un nuevo hijo del nodo en curso, este hijo pasa a ser el nodo en curso.

En Branch-and-Bound, 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).

En consecuencia, en Backtracking los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso y en Branch-and-Bound puede haber más nodos vivos que se almacenan en una lista de nodos vivos.

Por otro lado, en Backtracking, el test de comprobación realizado por las funciones de poda nos indica únicamente si un nodo concreto nos puede llevar a una solución o no. En Branch-and-Bound, sin embargo, se acota el valor de la solución a la que nos puede conducir un nodo concreto, de forma que esta acotación nos permite:

1. Podar el árbol si sabemos que no nos va a llevar a una solución mejor de la que ya tenemos.
2. Establecer el orden de ramificación de modo que comenzaremos explorando las ramas más prometedoras del árbol.

3. ESTUDIO DE LA IMPLEMENTACIÓN

El planteamiento general del problema a resolver es el siguiente:

Braulio es un trabajador que lleva toda una vida repartiendo los periódicos por todo el país. Todas las madrugadas, cuando la editorial cierra el diario y salen los primeros ejemplares de la imprenta, Braulio los reparte con su furgoneta entre todas las ciudades de EDAland.
Además de por su eficacia repartiendo, Braulio también destaca por ser una persona que tiene muy en cuenta su economía, y siempre procura escoger el circuito que le suponga recorrer la distancia más corta posible o realizar el menor gasto de combustible posible.
Para ser tan eficaz, Braulio se planteó tres cuestiones importantes.
La primera consistía en conocer todos los posibles circuitos, partiendo desde Almería, que recorren cada ciudad, donde debe dejar un lote de periódicos, exactamente una vez y regresando posteriormente a Almería.
La segunda cuestión consistía en determinar un circuito que recorra cada ciudad exactamente una vez, partiendo y regresando a Almería, y habiendo recorrido en total la menor distancia posible.
La tercera y última cuestión consistía en determinar un circuito que recorra cada ciudad exactamente una vez, saliendo y volviendo a Almería y cuyo gasto en combustible sea el mínimo posible.

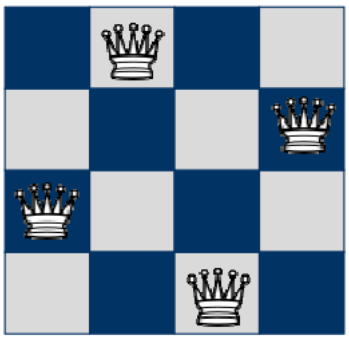
En esta práctica se deben proponer e implementar soluciones con los esquemas algorítmicos de backtracking y Branch-and-bound, según se requiera, a los problemas que se plantean a continuación.

3.1. Problema 1: Implementación con Backtracking.

Determinar todos los posibles circuitos, si es que hubiera más de uno, partiendo desde Almería, que recorren cada ciudad de la nueva red reducida de carreteras de EDAland, donde Braulio debe dejar un lote de periódicos, exactamente una vez y regresar a Almería. Si hubiera más de uno, indique entre todos ellos el circuito de menor distancia.

Implementar en Java el algoritmo *El problema del viajante* de las transparencias de clase de teoría, recomponiéndolo para que funcione y definiendo las variables de forma que se adapten a la recursividad.

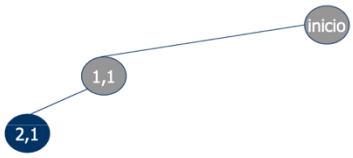
Antes de comenzar con la implementación, y para entender mejor como es el funcionamiento de este tipo de algoritmo, mostramos gráficamente un ejemplo de cómo se irían generando los estados en el problema de las 4 reinas utilizando backtracking:



Las restricciones básicas son las siguientes:

1.Todas las reinas deben estar en columnas diferentes.

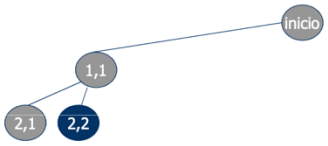
2.Todas las reinas deben estar en diagonales diferentes.

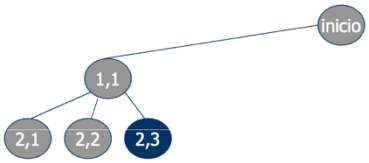


OK! Adelante con la búsqueda...

→

No se cumplen las restricciones: Misma columna.

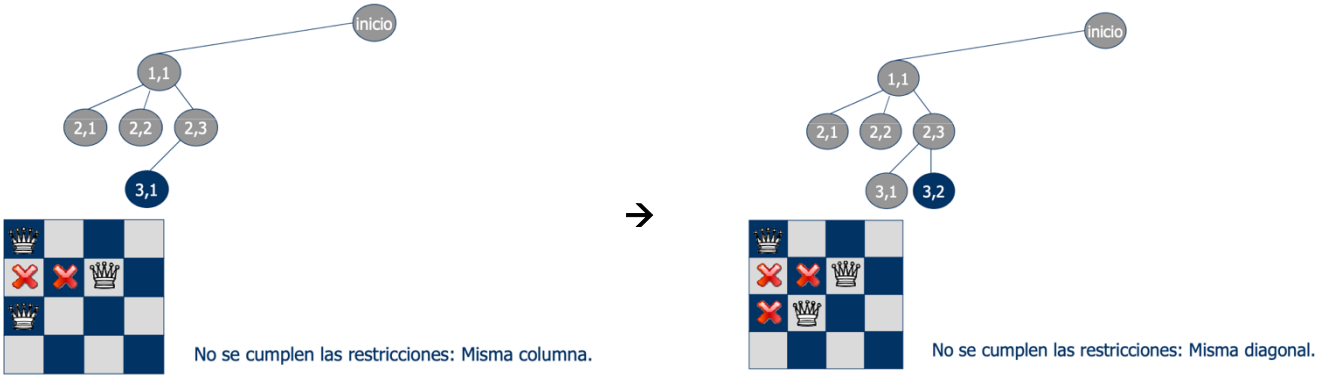




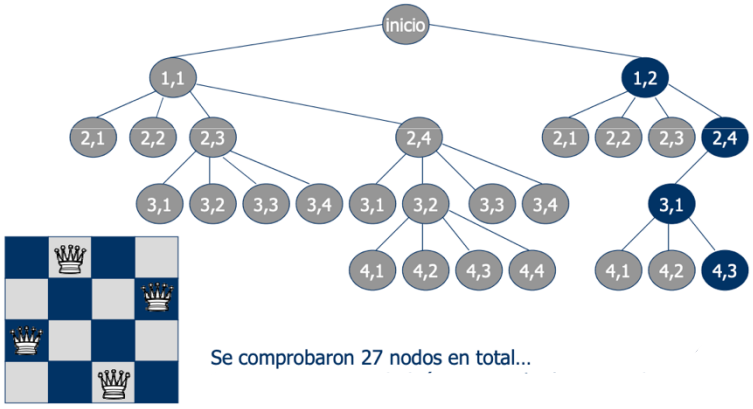
No se cumplen las restricciones: Misma diagonal.

→

OK! Adelante con la búsqueda...



Y así sucesivamente hasta encontrar la siguiente solución. Vemos como este proceso es similar al de una búsqueda en profundidad.



Una vez que conocemos de forma general el funcionamiento del algoritmo, lo primero que debemos plantear es qué estructuras vamos a usar. Inicialmente disponemos de un grafo en forma de mapa (HashMap).
Una vez revisado el esquema algorítmico general disponible en las diapositivas de clase, hemos pensado en tres estructuras que nos ayudarán a resolver el problema:

→ ***visited***: un mapa con clave el vértice y como valor un booleano que nos indicará si el vértice ha sido o no visitado.
→ ***results***: un ArrayList<PathNode>, siendo PathNode una clase que alberga la lista de vértices que compondrán un camino completo.
→ ***stage***: un array que va a ir almacenando cada uno de los vértices en cada etapa de profundidad del algoritmo en el árbol de exploración.

Además, usaremos una variable *Nivel* que indicará el nivel de profundidad en el que se encuentra nuestro algoritmo en cada etapa.

La primera estructura, **visited** (mapa de visitados), es preciso que lo rellenemos con todos los vértices que van a conformar el grafo y le asociemos un valor por defecto. En nuestro caso, hemos decidido usar un mapa que tiene como clave cada uno de los

vértices y como valor asociado un tipo booleano, inicialmente a false. El propósito de esta estructura es servirnos de guía para poder contemplar 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, tendremos que **r**ecorrer el conjunto de vértices del grafo e ir insertándolos en este mapa junto con un valor asociado a false.
Una vez lo hemos rellenado, es necesario pensar que nuestro punto de partida puede considerarse como visitado. Supongamos que partimos desde el vértice B, por lo que tendremos que establecer en ese mapa un valor de true.

En cuanto a la estructura **stage** (etapa), debemos establecer la primera y última posición con el valor del vértice desde el que partimos y al que llegamos respectivamente. Esta estructura inicialmente tendría valores a null para todas las posiciones, por lo que tenemos que sustituir la posición 0 y la última con el vértice de partida. La estructura tiene un total de N+1 posiciones siendo N el número total de vértices.

La estructura de **results** (resultados) en principio debe permanecer vacía, puesto que hasta que no se llame al algoritmo de Backtracking y se encuentren soluciones, no se agregarán.

Este es el método que inicializará las estructuras mencionadas:

```
private void initStructures(HashMap<String, Boolean> visited, String[] stage) {
    for (String v : graph.keySet()) {
        visited.put(v, false);
    }
    stage[0] = origin;
    stage[stage.length - 1] = origin;
    visited.put(origin, true);
}
```

Ahora que podemos inicializar las estructuras podemos llamar al algoritmo de Backtracking recursivo.
Cuando se implementa un algoritmo recursivo, es recomendable disponer de un método público, que suele recibir pocos parámetros, y por otra parte definir un método privado que recibe los parámetros definidos para que trabaje de forma recursiva.
El método público es similar al siguiente:

```
public ArrayList<ArrayList<String>> BackTracking() {
    HashMap<String, Boolean> visited = new HashMap<String, Boolean>();
    ArrayList<ArrayList<String>> results = new ArrayList<ArrayList<String>>();
    String[] stage = new String[numberOfVertex() + 1];
    int level = 1;

    initStructures(visited, stage);
    System.out.println("VISITED: " + visited);
    BackTracking(stage, level, visited, results);
    return results;
}
```

En cada llamada recursiva se intenten contemplar 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 hemos elegido nuestro nodo candidato, es necesario valorar si es factible 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 *stage* 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.

Sabremos que hemos llegado a una solución completa si cuando hayamos llegado a un nivel de profundidad igual al número de vértices menos 1 (Nivel == N-1). En tal caso, agregaremos la solución parcial (stage) a la estructura de soluciones (results).

En el caso de que la solución no sea completa, sabemos que debemos profundizar, por lo que haremos tres pasos:

1. Marcar el vértice candidato aceptado como visitado en nuestro mapa de visitados
2. Efectuar una llamada recursiva al algoritmo, teniendo en cuenta que en esta ocasión le pasamos el Nivel+1.
3. Cuando finalice la llamada recursiva, desmarcar como visitado el nodo candidato aceptado en el primer paso.

A continuación, puede verse el método recursivo de Backtracking utilizado:

```
private void BackTracking(String[] stage, int level, HashMap<String, Boolean> visited, ArrayList<ArrayList<String>> results) {
    for (Entry <String, Boolean> it : visited.entrySet()) { //SeleccionarNuevaOpcion hasta Ultima opción
        if(it.getValue()) continue; //Ya está visitado
        if(esAceptable(stage, level, it.getKey())) { //Si Aceptable
            stage[level] = it.getKey(); //Anotar opcion
            if(level == visited.size()-1) { //Si solución completa -> Incluir solucion
                ArrayList<String> temp = new ArrayList<String>(Arrays.asList(stage));
                double suma = 0;
                for(int i = 1; i < temp.size(); i++) {
                    double peso = getWeight(temp.get(i - 1), temp.get(i));
                    suma += peso;
                }
                temp.add("Coste: " + String.valueOf(suma));
                results.add(temp);
            }else {//Si solucion incompleta -> Etapa siguiente
                visited.put(it.getKey(), true);
                BackTracking(stage, level+1, visited, results);
                visited.put(it.getKey(), false);
            }
        }
    }
    stage[level] = null; //Marcar como no explorado
}
```

3.2. Problema 2. Implementación con Branch-and-Bound.

Determinar un circuito que, partiendo desde Almería, visite cada ciudad exactamente una vez, regresando a Almería y habiendo recorrido en total la menor distancia posible. Resolver este problema para la nueva red reducida de carreteras de EDAland.

Una de las particularidades que tiene *Branch & Bound* frente a *Backtracking* es la forma en la que se explora el árbol. Backtracking suele diseñarse para que realice una exploración en profundidad, buscando aquellas ramificaciones que den lugar a una solución, pero siempre dando prioridad a la profundización frente a la expansión.

Los algoritmos de Branch & Bound plantean otra forma de explorar los árboles, siendo esta en anchura. Si bien cabe pensar que al final puede llevar a lo mismo, en realidad no es cierto. El principal cometido que persiguen los algoritmos de ramificación y poda es priorizar las ramas candidatas con mayor probabilidad de éxito, es decir, intenta ir ramificando aquellas ramas sobre las que se estima un mayor éxito.

Para el ejercicio de B&B en esta práctica se ha propuesto hacer uso de dos plantillas disponibles en el repositorio de GitHub de la asignatura: una para ejecutar propiamente el algoritmo, y una segunda clase que actúa como estructura secundaria para crear objetos que representen un camino (Clase *PathNode*).

Respecto a esta clase *PathNode* comentar lo siguiente:

- Consta de un ArrayList donde se almacenarán de forma consecutiva los vértices que formarán el camino (*res*), un valor entero que indicará el número de vértices que se han visitado (*visitedVertex*), el coste total de las aristas que forman el camino actual (*totalCost*) y un valor de coste estimado para el camino (*estimatedCost*).
- El método compareTo será el encargado de dar preferencia a aquellos PathNodes que tengan un coste estimado más bajo.

A continuación, se puede observar cómo se ha completado la plantilla de la clase *PathNode* proporcionada:

```
public class PathNode extends TSP implements Comparable<PathNode> {

	private ArrayList<String> res;
	public int visitedVertex;
	public double totalCost;
	public double estimatedCost;

	public PathNode(String vertexToVisit) {
		res = new ArrayList();
		res.add(vertexToVisit);
		visitedVertex = 1;
		totalCost = 0.0;
		estimatedCost = numberOfVertex() * minEdgeValue;
	}
```

```
	public PathNode(PathNode parentPathNode) {
		this.res = new ArrayList(parentPathNode.res);
		this.visitedVertex = parentPathNode.visitedVertex;
		this.totalCost = parentPathNode.totalCost;
		this.estimatedCost = parentPathNode.estimatedCost;
	}

	@Override
	public int compareTo(PathNode p) {
		return Double.compare(this.estimatedCost,
p.estimatedCost);
	}

	public ArrayList getRes() {
		return this.res;
	}


	public void addVertexRes(String v) {
		this.res.add(v);
	}


	public String lastVertexRes() {
		return this.res.get(this.res.size() - 1);
	}


	public boolean isVertexVisited(String v) {
		return this.res.contains(v);
	}


	public int getVisitedVertices() {
		return visitedVertex;
	}


	public void setVisitedVertices(int visitedVertices) {
		this.visitedVertex = visitedVertices;
	}


	public double getTotalCost() {
		return this.totalCost;
	}


	public void setTotalCost(double totalCost) {
		this.totalCost = totalCost;
	}


	public double getEstimatedCost() {
		return estimatedCost;
	}

	public void setEstimatedCost(double estimatedCost) {
		this.estimatedCost = estimatedCost;
	}
```

Por último, y siguiendo la plantilla proporcionada, podemos ir definiendo como tiene que ser implementado el algoritmo Branch-and-Bound para este problema.

El primer método que desarrollaremos será uno nombrado como ***minimumEdgeValue,*** el cual se encargará de obtener el valor de arista mínimo de todo el grafo. El valor que se obtenga mediante este método será el que sirva para estimar qué ramificación puede resultar más interesante explorar primero.

A continuación, mostramos el método *minimunEdgeValue*, que se basa en recorrer el grafo y obtener el peso menor entre todas las aristas:

```
/**
 * Devuelve el valor minimo de la arista.
 * @return minimum - valor mínimo
 */
public double minimumEdgeValue() {
    double minimum = Double.MAX_VALUE;
    for (HashMap<String, Double> it : this.graph.values()) {
        for(Double value : it.values()) {
            if(value<minimum) {
                minimum = value;
            }
        }
    }
    return minimum;
}
```

Finalmente, y antes de mostrar el código global del algoritmo al que hemos llegado, comentamos de forma general cuál es su funcionamiento parte a parte:

1. Lo primero es definir la estructura auxiliar dónde se irán almacenando los caminos parciales. Se trata de una cola de prioridad de objetos PathNode, que permitirá mantener el conjunto de caminos ordenados de menor a mayor por su coste estimado.

   ```
   PriorityQueue<PathNode> priorityQueue = new PriorityQueue<PathNode>();
   ```

2. Después, deberemos definer la estructura que albergará el resultado final. En nuestro caso hemos utilizado un ArrayList de String.

   ```
   ArrayList<String> shortestCircuit = null;
   ```

3. Se definen algunas variables auxiliares y se inicializa la cola de prioridad con el primer camino, formado por el vértice de origen.

   ```
   int nVertices = numberOfVertex();
   minEdgeValue = minimumEdgeValue();
   double bestCost = Double.MAX_VALUE;
   PathNode firstNode = new PathNode(source);
   priorityQueue.add(firstNode);
   ```

4. Comienza el algoritmo iterativo de exploración. Comenzará con un bucle condicionado por el tamaño de la cola de prioridad, de modo que mientras existan elementos dentro de la cola, se permanecerá dentro de él. El primer paso es extraer el primer PathNode de la cola, siendo el que menos coste estimado tiene y evaluar si su coste estimado es aceptable, es decir, si su coste estimado es inferior al mejor coste de solución obtenido hasta el momento. En caso de no serlo, simplemente se descarta.

```
while(priorityQueue.size() > 0) {
        PathNode Y = priorityQueue.poll();
        if (Y.getEstimatedCost() >= bestCost) break;
```

5. Si pasa el filtro, el siguiente paso es extraer el último vértice del PathNode (from) que extrajimos anteriormente y, seguidamente determinar si el camino actual es una solución completa o no lo es. Esto lo sabremos cuando el número de vértices del PathNode sea el mismo que el número de vértices y que, además, ese último vértice from tiene una arista que lo una al vértice de partida.
   Entonces, si es un camino completo solo queda determinar si el coste total (incluyendo la arista desde from hasta el punto de partida) es mejor que el valor de solución actual, en cuyo caso se establecerá como la nueva mejor solución.

```
String from = Y.lastVertexRes();
if ((Y.getVisitedVertices() == numberOfVertex()) && (containsEdge(from, source))) {
        Y.addVertexRes(source);
        Y.totalCost += getWeight(from,  origin);
        if (Y.getTotalCost() < bestCost) {
                bestCost = Y.getTotalCost();
                shortestCircuit = new ArrayList<String>(Y.getRes());
        }
```

6. Si por el contrario se determina que no es un camino completo, será necesario seguir profundizando, buscando entre los vértices adyacentes que no estén visitados. En ese caso, se crea un PathNode copia y se le agrega el nuevo vértice adyacente no visitado, se incrementa el número de vértices visitados, se incrementa el coste total con el peso de la arista y se recalcula el coste estimado.

```
} else {
        for(String to : getNeighbors(from)){
                if (!Y.isVertexVisited(to)) {
                        PathNode X = new PathNode(Y);
                        X.addVertexRes(to);
                        X.visitedVertex++;
                        X.totalCost += getWeight(from, to);
                        X.estimatedCost = X.totalCost + (nVertices - X.getVisitedVertices() + 1)
                * minEdgeValue;
```

```
    public ArrayList<String> TSPBaB(String source) {
        if (graph.get(source) == null)
            return null;

        PriorityQueue<PathNode> priorityQueue = new PriorityQueue<PathNode>();
        ArrayList<String> shortestCircuit = null; //El mejor camino encontrado
        int nVertices = numberOfVertex();
        minEdgeValue = minimumEdgeValue();
        double bestCost = Double.MAX_VALUE;
        PathNode firstNode = new PathNode(source); // Constructor de clase PathNode
        priorityQueue.add(firstNode);

        while(priorityQueue.size() > 0) {
            PathNode Y = priorityQueue.poll(); // Y = menorElemento de la cola de prioridad en funcion de 'estimatedCost'
            if (Y.getEstimatedCost() >= bestCost) break;
            String from = Y.lastVertexRes();
                // Si el numero de vertices visitados es n
                // y existe una arista que conecte 'from' con source
                if ((Y.getVisitedVertices() == numberOfVertex()) && (containsEdge(from, source))) {
                    Y.addVertexRes(source);// Actualizar 'res' en Y añadiendo el vertice 'source'
                    Y.totalCost += getWeight(from,  origin);// Actualizar 'totalCost' en Y.
                    if (Y.getTotalCost() < bestCost) {
                        bestCost = Y.getTotalCost();
                        shortestCircuit = new ArrayList<String>(Y.getRes());
                    }
                } else {
                    for(String to : getNeighbors(from)){ // Iterar para todos los vertices advacentes a from.
                        if (!Y.isVertexVisited(to)) { //Si el vertice 'to' todavia no ha sido visitado en Y.
                            PathNode X = new PathNode(Y); // Creamos un pathnode nuevo 'X' (copia de 'Y').
                            X.addVertexRes(to);// Anadir 'to' a 'res' en X
                            X.visitedVertex++; // Incrementar en 1 los vertices visitados en X
                            X.totalCost += getWeight(from, to);// Actualizar 'totalCost' en X.
                            X.estimatedCost = X.totalCost + (nVertices - X.getVisitedVertices() + 1) * minEdgeValue; // Actualizar
                            if (X.getEstimatedCost() < bestCost) {
                                priorityQueue.add(X);
                            }
                        }
                    }
                }
            }
        return shortestCircuit;
    }

}
```

4. ANÁLISIS DE EFICIENCIA.

Backtracking

De los múltiples factores que afectan a la eficiencia de backtracking el número de nodos generados juega un papel fundamental.
En el **mejor caso**, backtracking encuentra la solución en el primer recorrido en profundidad, por tanto, solo se tienen que analizar n nodos y su orden de complejidad seria O(n).

En el **peor de los casos**, se ha de explorar el árbol completo y por tanto se puede alcanzar un orden exponencial $O(2^n)$ o factorial O(n!).

En cuando a la eficiencia espacial, los algoritmos con backtracking tienen asociados unos requisitos de memoria, propia del esquema backtracking, O(n), dados por la máxima profundidad de las llamadas recursivas.

Branch-and-Bound

La técnica Branch-and-Bound da lugar a algoritmos de complejidad exponencial, por lo que normalmente se utiliza en problemas complejos que no pueden resolverse en tiempo polinómico (NP- completos).
El tiempo de ejecución de un algoritmo "Branch & Bound" depende de:

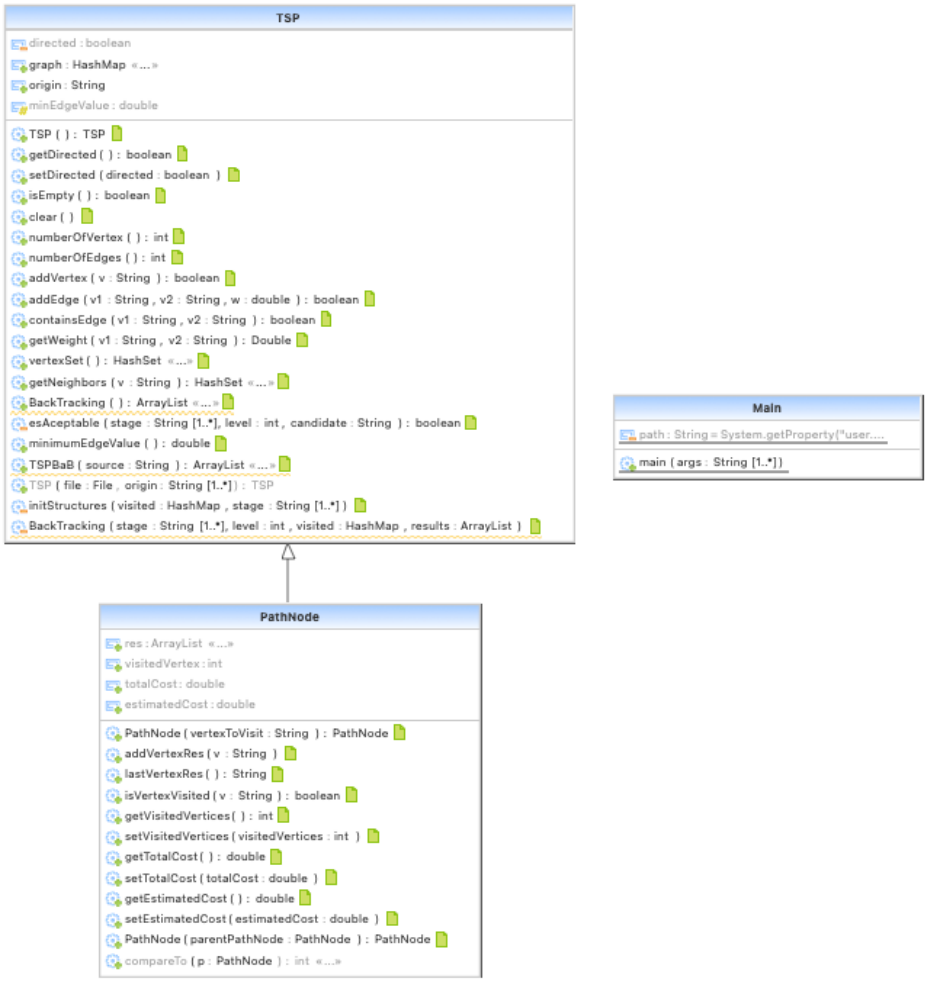
- El número de nodos recorridos (que, a su vez, depende de la efectividad de la poda).
- El tiempo empleado en cada nodo (tiempo necesario para hacer las estimaciones de coste y gestionar la lista de nodos vivos en función de la estrategia de ramificación).

En el **peor caso**, el tiempo de un algoritmo Branch-and-Bound" será igual al de un algoritmo backtracking, **exponencial** (o peor incluso, si tenemos en cuenta el tiempo que requiere el cálculo de cotas y la gestión de la lista de nodos vivos).

En el **caso promedio**, no obstante, se suelen obtener mejoras con respecto a backtracking, dependiendo de las funciones de poda.

En cuanto a la **eficiencia espacial**, los requerimientos de memoria son mayores que los de los algoritmos con backtracking.
Ya no se puede disponer de una estructura global donde ir construyendo la solución, puesto que el proceso de construcción no es tan sistemático como antes → **anotación de los recorridos más compleja**.
Ahora, cada nodo debe ser autónomo, en el sentido de contener toda la información necesaria para la ramificación y la poda, y para reconstruir la solución encontrada hasta ese momento → **aumenta el costo en memoria de cada nodo**.
Los nodos vivos pueden ser más de n, siendo n la profundidad del árbol (número mayor que con Bactracking) → **Aumenta el número de nodos.**

5.ANEXO – Diagrama de Clases



6. Fuentes

[1] Algoritmos Bactracking – Universidad Don Bosco, disponible en https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/programacion-iv/2019/ii/guia-11.pdf

[2] Exploración de grafos, algoritmos Branch & Bound – Universidad de Granada, disponible en https://elvex.ugr.es/decsai/algorithms/slides/5%20Branch%20and%20Bound.pdf