Juan Raul Mellado Garcia y Alejandro Martinez Terrones

Práctica 4 EDA II

Algoritmo Backtracking

Introducción Backtracking

La estrategia del Backtracking o vuelta es una estrategia en la que buscamos una solución que cumpla unas condiciones específicas, a diferencias de otras técnicas o estrategias no se prioriza tanto la velocidad, si no el poder encontrar las condiciones óptimas.

El funcionamiento de esta técnica es el de desarrollar todas las soluciones y elegir la más conveniente para nuestro caso.

Tenemos la opción para hacerlo de manera más óptima, de cuando una rama de solución no cumple los requisitos se descarta y se vuelva al punto en el que si cumplia las condiciones para tomar otro camino, evitando así cálculos inútiles y mejorando el tiempo de ejecución, a esta técnica se le llama ramificación y poda (branch and bound) .

BackTracking

```
public ArrayList<List<String>> viajero() {

    soluciones = new ArrayList<>();
    TreeMap<String, Double> info = null;

    String[] camino = new String[nodos.length + 2];
    for (int i = 0; i < camino.length; i++)
        camino[i] = null;

    camino[0] = nodos[0];
    viajeroRecursivo(0, 0, camino, info);

    return soluciones;

}
```

Nuestro algoritmo es recursivo y aquí tenemos el método público de este, como podemos ver, lo primero que hacer es inicializar soluciones, que es un ArrayList que va a contener en su interior todas las soluciones que cumplan con nuestras restricciones.

Info a su vez es un TreeMap que vamos a usar recursivamente para tener los nodos conectados al actual y el coste de estos.

Y por último creamos caminos , que es un array de string que se encargará de tener en su interior los nodos recorridos y en el último espacio la distancia recorrida, por esto mismo su tamaño es el tamaño de los nodos más 2, ya que hay que pasar 2 veces por Almería y el otro espacio es para la distancia.

```java
private void viajeroRecursivo(int posicion, double distancia, String[] caminoFinal,
        TreeMap<String, Double> info) {

    if (caminoFinal[nodos.length - 1] != null) {

        info = this.net.getAdjacencyMap().get(caminoFinal[nodos.length - 1]);
        if (info.containsKey(caminoFinal[0])) {
            caminoFinal[nodos.length] = caminoFinal[0];
            double miDistancia = info.get(caminoFinal[0]) + distancia;
            caminoFinal[nodos.length + 1] = String.valueOf(miDistancia);

            soluciones.add(new ArrayList<String>(Arrays.asList(caminoFinal)));
        }


    } else {

        for (int i = 0; i < nodos.length; i++) {

            for (int x = posicion + 1; x < caminoFinal.length; x++)
                caminoFinal[x] = null;
            info = this.net.getAdjacencyMap().get(caminoFinal[posicion]);

             if (!Arrays.asList(caminoFinal).contains(nodos[i]) && info.containsKey(nodos[i])) {

                caminoFinal[posicion + 1] = nodos[i];

                double miDistancia = info.get(nodos[i]) + distancia;

                viajeroRecursivo(posicion + 1, miDistancia, caminoFinal, info);

            }

        }

    }

}
```

Una vez dentro del método recursivo podemos ver la comprobación de si el último hueco está lleno lo que significaba el final de esa rama, teniendo en ese if la comprobación de si el último es almería y añadiendolo a la solución con la distancia recorrida.

Por otro lado el else significa que no es es el final de la rama, por lo cual rellena info con los nodos adyacentes al actual y si alguno de esos nodos no está recorrido en el camino lo añade y vuelva a hacer la llamada recursiva.

Branch and bound

```java
@SuppressWarnings("unchecked")
public ArrayList<Vertex> TSPBaB(Vertex source) {
    TreeMap<Vertex, Double> neighborMap = adjacencyMap.get(source);
    if (neighborMap == null)
        return null;

    minEdgeValue = minimumEdgeValue();

    // Constructor de clase PathNode
    PathNode firstNode = new PathNode(source);

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

    priorityQueue.add(firstNode);

    shortestCircuit = null;
    double bestCost = Double.MAX_VALUE;
    double costeTotal = 0;

    while(priorityQueue.size() > 0) {
        // Y (PathNode) = menorElemento de la cola de prioridad en funcion de 'estimatedCost'
        PathNode Y = priorityQueue.poll();
        if (Y.getEstimatedCost() >= bestCost)
            break;
        else {
            Vertex from = Y.lastVertexRes();
            // Si el numero de vertices visitados es n
            // y existe una arista que conecte 'from' con source
            if ((Y.getVisitedVertices() == numberOfVertices()) &&
                (containsEdge(from, source))) {
                // Actualizar 'res' en Y añadiendo el vertice 'source'
                // Actualizar 'totalCost' en Y con Y.totalCost + weight(from, source)
                Y.res.add(source);
                Y.totalCost+=this.getWeight(from, source);
                if (Y.getTotalCost() < bestCost) {
                    // Actualizar 'bestCost', 'shortestDistanceCircuit' y 'shortestCircuit'
                    bestCost=Y.totalCost;
                    shortestCircuit=Y.res;
                    //shortestDistanceCircuit??????????

                }
            }
```

Como vemos en este caso lo primero que hacemos es inicializar lo que necesitaremos, un TreeMap, que usaremos para los nodos adyacentes, la arista de menor valor del grafo, una instancia de la clase PathNode, la PQ que va a llevar dentro los nodos a visita y costes total y best cost para llevar un seguimiento de las ramas.

Nuestro while va a seguir dando iteraciones hasta que se rompa la ejecución desde dentro o se que vacía la PQ.

Lo primero que hacemos es añadir al camino al próximo nodo de menos coste, si el coste estimado del camino es mayor al mejor corte se corta la ejecución, en caso contrario

se comprueba si el número de vértices visitados es n y si existe una arista que conecte ambos, si es asi se añade al camino y si su coste total es mas optimo que el mejor coste total actual , se iguala al recorrido más corto.

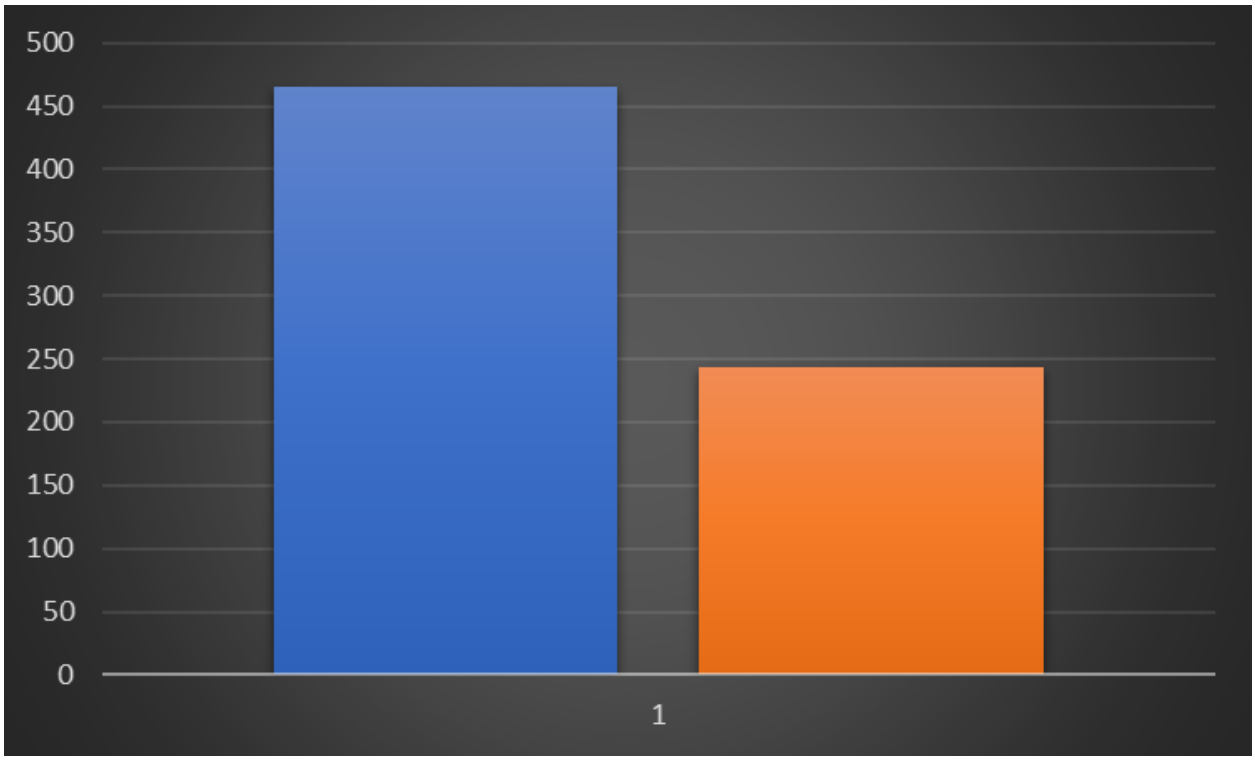
```
                }
                else {
                    // Iterar para todos los vertices advacentes a from,
                    // a cada vertice lo denominamos 'to'
                    for(Vertex to : adjacencyMap.get(from).keySet()) {
                        //if (vertice 'to' todavia no ha sido visitado en Y) { // hacer uso de la funcion 'isVertexVisited(vertex)' de PathNode
                        if(!Y.isVertexVisited(to)) {
                            PathNode X = new PathNode(Y); // Uso de constructor copia
                            // Anadir 'to' a 'res' en X
                            X.res.add(to);
                            // Incrementar en 1 los vertices visitados en X
                            X.visitedVertices++;
                            // Actualizar 'totalCost' en X con Y.totalCost + weight(from, to)
                            X.totalCost+=this.getWeight(from, to);
                            // Actualizar 'estimatedCost' en X con X.totalCost + ((nVertices - X.getVisitedVertices() + 1) * minEdgeValue)
                            X.estimatedCost+=((X.visitedVertices - X.getVisitedVertices() + 1))*minEdgeValue;

                            if (X.getEstimatedCost() < bestCost) {
                                priorityQueue.add(X);
                                //System.out.println("Hola");
                            }
                        }
                    }
                }
            }
            //System.out.println(Y.totalCost);
            costeTotal=Y.totalCost;
        }
        shortestCircuit.add((Vertex)String.valueOf(costeTotal));
        //System.out.println(shortestCircuit);
        return shortestCircuit;
    }
```

En caso de que el número de vértices visitados no sea n , se llega a este fragmento de código, que lo que hace es añadir al camino el siguiente vértice, y actualizar lo valores del camino en función del nodo que se le acaba de añadir y si este nodo no supera el coste estimado se sigue explorando ese camino.

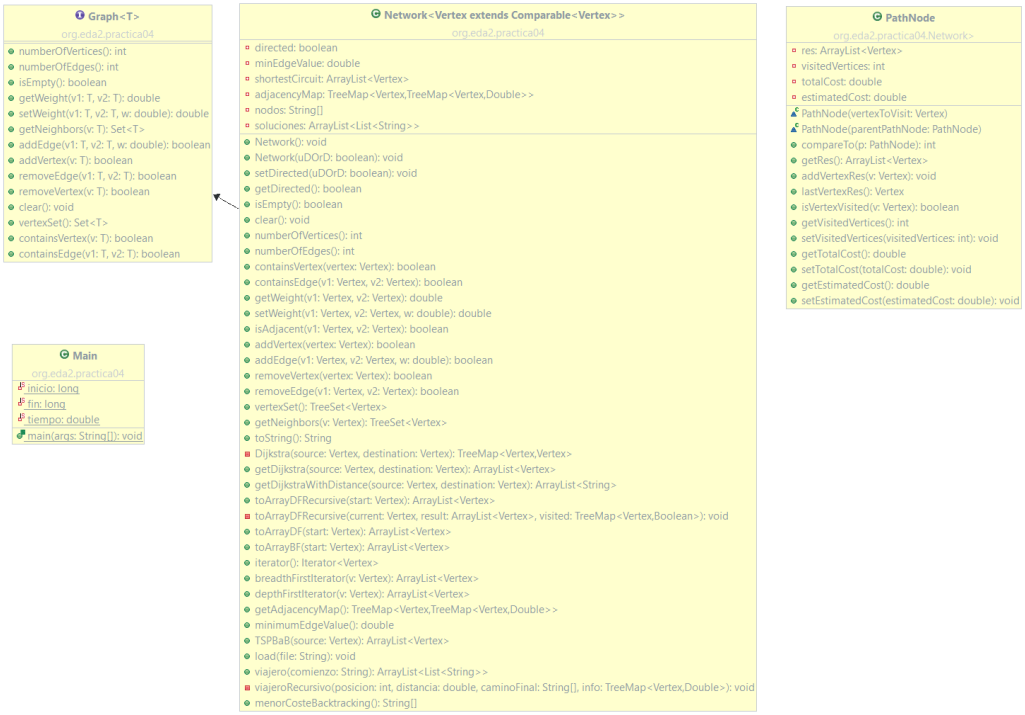
+

Mapa EDALAND



Aquí tenemos la gráfica sacada con el tiempo medio de 10 ejecuciones, la izquierda es BackTracking con un tiempo medio de 465,4 ms y a la izquierda la rama y poda con un tiempo medio de 243,8 ms , como ya cabía esperar teóricamente la rama y poda es mas rapida ya que corta las rama cuando detecta que no van a ser óptimas o que no se ajusta a lo que buscamos mientra que el backtracking por su lado, la única mejora que le hemos implementado es que no repita nodos ya visitados evitando asi gasta mucho tiempo en bucles.

Anexo



Bibliografía

Para la realización de esta práctica solo se ha consultado los apuntes de la asignatura.