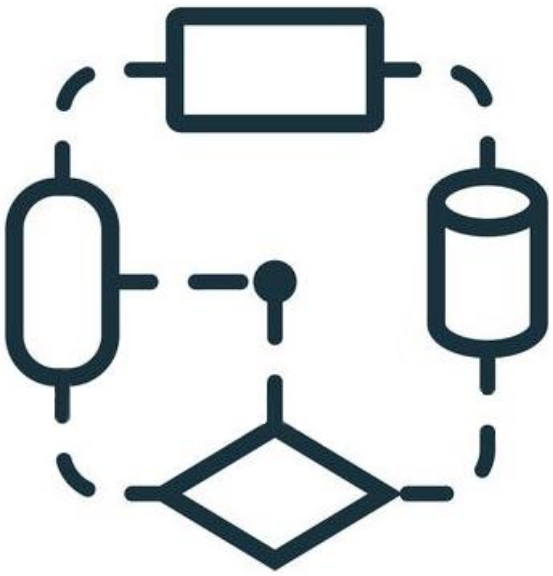
Backtracking

PRÁCTICA 4



Autor: Pablo Gómez Rivas y Jesús Fornieles Muñoz

Materia: Estructura de Datos y Algoritmos II

Grupo de Prácticas: GTAB2

Fecha: Jueves, 18 de mayo de 2023

Índice

Objetivo **3**
Antecedentes **4**
Estudio de la Implementación **5**
Todos los posibles circuitos hamiltonianos. 5
Circuito hamiltoniano de menor recorrido. 7
Circuito hamiltoniano de menor coste en gasolina. 7
Añadir vértices al grafo de EDALand 8
Estudio Teórico **8**
Estudio Experimental **11**
TSPBacktrackingTodos O(n!) 11
TSPBacktrackingMejor O(n!) 11
Apartado 5 O(n!) 12
Anexo **13**
Archivos fuente: 14
Fuentes Bibliográficas **15**

Objetivo

El backtracking, o vuelta atrás en español, es una técnica utilizada en programación para buscar todas las soluciones posibles a un problema mediante una exploración exhaustiva de un árbol de decisiones.

Consiste en construir incrementalmente una solución parcial, verificando si cumple ciertas condiciones y, si no las cumple, retroceder (backtrack) para probar una opción alternativa. De esta manera, se exploran todas las posibilidades hasta encontrar la solución óptima o todas las soluciones válidas.

El proceso de backtracking se basa en la recursión y sigue los siguientes pasos generales:

- Definir el espacio de soluciones: Se debe establecer una representación del problema como un árbol o un grafo, donde cada nodo representa una elección o decisión que se puede tomar para llegar a una solución.

- Definir una función recursiva: Se crea una función que realiza la exploración del árbol de decisiones, construyendo incrementalmente la solución parcial.

- Verificar las condiciones de terminación: Se definen las condiciones de terminación para determinar cuándo una solución parcial es válida o cuándo se ha alcanzado la solución final.

- Verificar las condiciones de poda: Se incluyen condiciones adicionales para evitar explorar subárboles que no contienen soluciones válidas. Esto ayuda a reducir el espacio de búsqueda y mejorar la eficiencia del algoritmo.

- Retroceder (backtrack) y probar opciones alternativas: Si una solución parcial no cumple con las condiciones requeridas o no se puede continuar, se retrocede a un nivel anterior y se prueba una opción alternativa.
-

El proceso de backtracking se repite hasta agotar todas las opciones posibles y encontrar la(s) solución(es) deseada(s). Es importante tener en cuenta que el backtracking puede ser ineficiente para problemas grandes debido a su naturaleza exponencial en el peor de los casos. Por lo tanto, es importante aplicar técnicas de poda y optimización para mejorar su eficiencia.

En resumen, el backtracking es una técnica poderosa para buscar todas las soluciones posibles de un problema al explorar exhaustivamente un espacio de soluciones utilizando recursión y retroceso para encontrar la solución óptima o todas las soluciones válidas.

En esta ocasión vamos a hacer uso de esta metodología de programación con el fin de ayudar a Marcial el comercial, el mejor empleado de la empresa EDA-Cola, cuya sede se encuentra en Almería. Todos los inicios de mes Marcial realiza un viaje a las ciudades más importantes de EDALand para cerrar los pedidos del mes. De este forma nace en Marcial la necesidad de tratar de realizar el camino más corto (y menos costoso en gasolina) para llegar a todas las ciudades sin dejarse ninguna.

Para conseguir esta misión se plantean 3 preguntas que resolveremos durante el apartado de antecedentes.

Por otro lado, en lo que respecta a este trabajo, hemos decidido hacer a Pablo Gómez Rivas líder de esta práctica, encargado de supervisar y verificar la calidad y el correcto funcionamiento de los códigos del trabajo del resto de miembros, Jesús Fornieles Muñoz en este caso, de forma que la elaboración del proyecto ha quedado de la siguiente forma:

#	Tipo	Tarea	
1	Implementacion	**Algoritmo todos los circuitos posibles**	Pablo
2	Implementacion	**Algoritmo circuito menor distancia posible**	Jesús
3	Implementacion	**Algoritmo combustible**	Pablo
4	Implementacion	**Algoritmo cualquier ciudad de origen**	Jesús
5	Implementacion	**Algoritmo añadir nuevas capitales**	Pablo
6	Implementacion	Generación de secuencuas aleatorias. Considerar casos especiales	Jesús
7	Implementacion	Main con medidas de tiempos de ejecución	Pablo
8	Implementacion	Juegos de prueba, considerando los casos especiales	Jesús
9	Documentación	**Estudio de la implementación**	Pablo
10	Documentación	**Estudio teórico**	Jesús
11	Documentación	**Estudio experimental**	Pablo
12	Documentación	**Anexos** y bibliografía	Jesús

Antecedentes

Durante la lectura del informe nos damos cuenta de que el problema que tiene marcial se asemeja considerablemente a un problema recurrente en los algoritmos relacionados con el backtracking, el problema del del viajante (TSP). Este se basa en la teoría de Hamilton de grafos, con el fin de encontrar el ciclo de Hamilton más corto del grafo.

Ahora bien, las preguntas que plantea Marcial son las siguientes:

- La primera consiste en conocer todos los posibles circuitos, partiendo desde Almería, que pasan por cada ciudad de EDAland, donde negociar con el responsable de bebidas, exactamente una vez y regresando posteriormente a Almería.

- La segunda cuestión consiste en determinar un circuito que pase por cada ciudad exactamente una vez, partiendo y regresando a Almería, y habiendo recorrido en total la menor distancia posible.

- La tercera y última cuestión consiste en determinar un circuito que pase por cada ciudad exactamente una vez, saliendo y volviendo a Almería y cuyo gasto en combustible sea el mínimo posible, sabiendo que suele repostar siempre en las mismas gasolineras (ciudades) donde no siempre hay un precio unificado de gasolina.

Ahora bien, estas preguntas de momento son un auténtico misterio en este momento, pero durante el desarrollo del informe veremos cómo aparecen distintos algoritmos que nos van a permitir darle respuesta a estas incógnitas.

Estudio de la Implementación

Todos los posibles circuitos hamiltonianos.

En este algoritmo hayamos la respuesta a la primera pregunta que se plantea Marcial, cuales son todos los posibles caminos que empiezan en Almería, visitan el resto de ciudades una sola vez y vuelven a Almería.

Para desarrollar este algoritmo hacemos una serie de modificaciones en el algoritmo btSalespersonBest, el cual encuentra el ciclo de menor coste. Ahora bien, dado que para encontrar el mejor ciclo es necesario recorrer todos los ciclos posibles, hacemos uso de este mecanismo para guardar en una variable auxiliar de la clase todos los caminos posibles (y sus correspondientes espejos).

En cuanto al código, comenzamos en un método pre-recursivo en el cual inicializamos una serie de variables necesarias para el algoritmo backtracking .

```java
public static ArrayList<Double> btSalesperson(double[][] G) {
    // set partialTour to identity permutation
    int n = G.length;
    partialTour = new int[n];
    for (int i = 0; i < n; i++)
        partialTour[i] = i;

    allTours = new ArrayList<>();
    costOfAllTours = new ArrayList<>();
    costOfPartialTour = 0;

    // search permutations of partialTour[2:n]
    rTSP(G, 1, n);
    return costOfAllTours;
}
```

Después de la llamada recursiva, ya habremos modificado nuestro atributo auxiliar con todos y cada uno de los posibles ciclos:

```java
static ArrayList<int[]> allTours;
```

Por otro lado el método devuelve un array de costes asignados a cada uno de los recorridos de la variable allTours.

En cuanto a la recursividad encontramos el siguiente método:

```java
public static void rTSP(double[][] G, int currentLevel, int n) { // search from a node at currentL
    if (currentLevel == n) { // at parent of a leaf
        if (G[partialTour[n - 1]][0] != 0) {
            int[] aux = new int[partialTour.length];
            System.arraycopy(partialTour, 0, aux, 0, partialTour.length);
            allTours.add(aux);
            costOfAllTours.add(costOfPartialTour + G[partialTour[n - 1]][0]);
        }
    } else { // try out subtrees
        for (int j = currentLevel; j < n; j++) {
            // is move to subtree labeled partialTour[j] possible?
            if (G[partialTour[currentLevel - 1]][partialTour[j]] != 0) {
                swap(partialTour, currentLevel, j);
                costOfPartialTour += G[partialTour[currentLevel - 1]][partialTour[currentLevel]];
                rTSP(G, currentLevel + 1, n);
                costOfPartialTour -= G[partialTour[currentLevel - 1]][partialTour[currentLevel]];
                swap(partialTour, currentLevel, j);
            }
```

Como vemos la recursividad se divide en dos partes bien diferenciadas:

Current level == n, esto implica que hemos llegado a un nodo padre de una hoja del árbol de búsqueda, por lo que nuestro siguiente paso es comprobar si podemos completar el tour.

En esta parte se verifica si el último nodo del recorrido parcial partialTour está conectado al nodo inicial.

Ahora bien, dentro de este condicional determinamos si el ultimo nodo del recorrido parcial esta unido o no con el nodo fuente, y en caso de

que lo esté añadimos tanto el recorrido completo a nuestra lista de recorridos posibles como a nuestra lista de coste de cada camino.

Por el otro lado, en caso de que n != currentLevel significa que tenemos que seguir explorando subárboles donde a partir de un bucle junto con la llamada recursiva prosigue la expansión de subárboles hasta dar con un nodo hoja.

Circuito hamiltoniano de menor recorrido.

En este algoritmo respondemos a la segunda pregunta de Marcial, donde se cuestiona cual sería el camino más corto de los que hemos conseguido en el algoritmo anterior.

Realmente la estructura de este algoritmo resulta muy similar a la del algoritmo anterior debido a que en este caso volvemos a recorrer todos los caminos posibles solo que para satisfacer este método es necesario obtener el mejor resultado. Para ello cuando en la recursividad n == current level (llegamos a nodo hoja) guardamos la solución obtenida solo si cumple estas dos condiciones:

- El ultimo nodo del recorrido parcial esta conectado con nuestro nodo fuente.
- El coste del recorrido obtenido es menor a la iteración anterior, es decir que ha mejorado el anterior camino.

```
if (G[partialTour[n - 1]][0] != 0 && (costOfPartialTour + G[partialTour[n - 1]][0] < costOfBestTourSoFar))
```

En el caso en el que n != current level el código resulta prácticamente idéntico al del algoritmo anterior solo que en este volvemos a realizar comparaciones para ver si estamos mejorando o no el caso anterior.

Circuito hamiltoniano de menor coste en gasolina.

En este algoritmo damos con la solución a la tercera pregunta que formula Marcial. Debido a su afán ahorrador, nos vemos en la obligación de recalcular el resultado que nos ofrece el algoritmo anterior para conseguir ahora aquel cuyo coste de gasolina sea el mínimo posible.

Este cambio reside en que las aristas ahora obtendrán nuevos valores al ser multiplicadas su valor por un numero aleatorio entre 0,05 y 0,1. Esto supone la necesidad de un nuevo cálculo del camino mínimo ahora aplicado al nuevo grafo.

Para ello aplicamos el algoritmo del apartado anterior de nuevo contemplando que pueden haber cambios en el camino que obtuvimos antes, ya que el valor de las aristas ha variado.

Añadir vértices al grafo de EDALand

Este algoritmo nos permite satisfacer algunas de la funcionalidades que se nos exigen en los apartados 4 y 5. En concreto con este método somos capaces de añadir una serie de nodos al grafo de EDALand y conectarlos con otros nodos distintos. Esto nos otorga un mecanismo para comparar tiempos de ejecución en función del número de vértices (n), ya que al ser de orden factorial (lo veremos más adelante) aumenta considerablemente al añadir un nuevo nodo.

Para añadir los vértices y las aristas hacemos uso de un string (nodo a añadir) treemap de aristas que incluya el destino y el peso de la arista.

```
public static boolean addCapitalProvincia(String nombre, TreeMap<String, Integer> aristas)
```

El mecanismo para añadir núcleos consiste en los siguientes pasos:

- Crear una copia del archivo original
- Comprobar si ya existe el vértice que queremos introducir
- Volver a copiar el archivo solo que añadiendo el vértice y sus nuevas aristas.

Estudio Teórico

Los tres métodos presentados tienen el mismo orden de complejidad ***O*(n!)** porque comparten la misma estructura básica de algoritmo. Esto es debido a que mediante el mecanismo de backtracking podemos acceder a diversos datos que en pueden satisfacer cada uno de los apartados de los cuales queremos hacer el estudio teórico.

Dada esta similitud, vamos a presentar solamente el estudio teórico del algoritmo que encuentra el mejor ciclo Hamiltoniano (btSalespersonBest), por ejemplo:

Método no recursivo:

```
    public static double btSalespersonBest(double[][] G,
int src) {
        // set partialTour to identity permutation
        int n = G.length;
        partialTour = new int[n];
        for (int i = 0; i < n; i++)
            partialTour[i] = i;

        for (int i = 0; i < n; i++){
```

```
            double maxCost = 0;
            boolean outEdge = false;
            for (int j = 0; j < n; j++){

                if (G[i][j] != 0)
                    outEdge = true;
                    if (G[i][j] > maxCost)
                        maxCost = G[i][j];
                    }
                }bestTourSoFar = new int[n];
            costOfPartialTour = 0;

            // search permutations of partialTour[2:n]
            rTSPBest(G, 1, n);
            return costOfBestTourSoFar;
        }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante.
- La parte verde marca la llamada al método recursivo.

$$\sum_{i=0}^{n}(c) + \sum_{i=0}^{n}(\sum_{j=0}^{n}(c)) \in O(n^2)$$

- Método recursivo:

```
public static void rTSPBest(double[][] G, int currentLevel, int n) {
    if (currentLevel == n) {
        if (G[partialTour[n - 1]][0] != 0 && (costOfPartialTour +
        G[partialTour[n - 1]][0] < costOfBestTourSoFar)) {
        for (int i = 0; i < n; i++)
        bestTourSoFar[j] = partialTour[j];
        costOfBestTourSoFar = costOfPartialTour +
    G[partialTour[n - 1]][0];
            }
    } else {
        for (int j = currentlevel; j < n; j++){
            if (G[partialTour[currentLevel - 1]][partialTour[j]]
            != 0 && (costOfPartialTour +
            G[partialTour[currentLevel - 1]][partialTour[j]] <
            costOfBestTourSoFar)) {
            swap(partialTour, currentLevel, j);
```

```
                costOfPartialTour += G[partialTour[currentLevel -
                1]][partialTour[currentLevel]];
                double estimatedTourCost = costOfPartialTour + ((n -
                currentLevel + 1) * costMinimumEdge);
                if (estimatedTourCost < costOfBestTourSoFar)
                        rTSPBest(G, currentLevel + 1, n);
                costOfPartialTour -= G[partialTour[currentLevel -
                1]][partialTour[currentLevel]];
                swap(partialTour, currentLevel, j);
                                        }
                                }
                        }
                }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante.
  En este caso nos encontramos un caso especial dada la recursividad dentro del bucle for. Como hemos visto en clase se trata de un caso concreto de

El bucle principal rTSP es recursivo y se ejecuta para cada nivel del árbol de permutaciones. Cada nivel se ejecuta una vez por cada posible valor de j en el bucle for.

La suma total de las iteraciones del bucle **for** en el bucle principal es (n-1) + (n-2) + (n-3) + ... + 1, lo cual es equivalente a (n-1)*n/2.

Simplificando la expresión, obtenemos una complejidad de O(n^2) para el bucle principal rTSP.

En el peor caso, currentLevel puede tomar valores desde 1 hasta n-1, y j puede tomar valores desde currentLevel hasta n-1. Por lo tanto, el bucle principal tiene una complejidad de O(n!) en el peor caso.

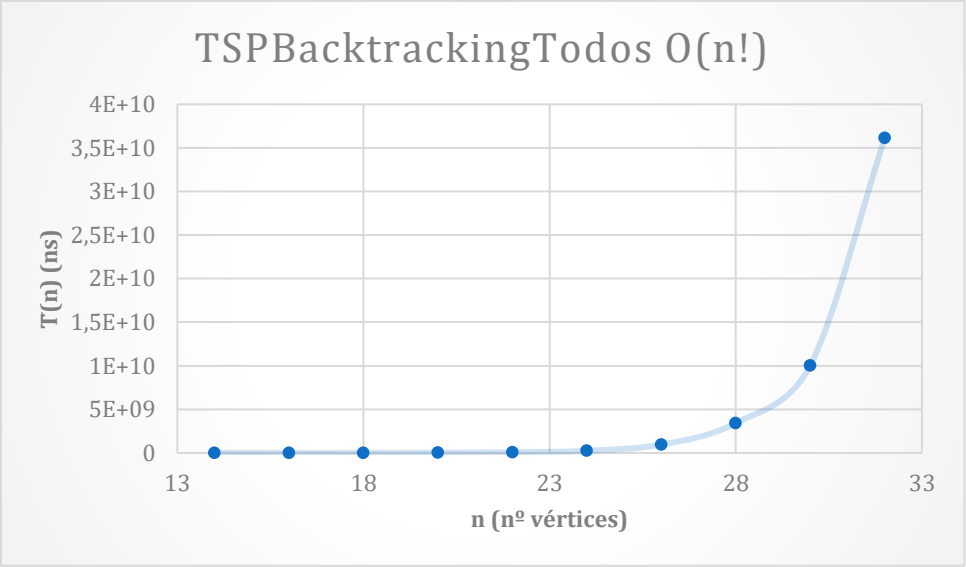
Dado que al ser Big O, cogemos siempre el peor caso, es decir, la complejidad del algoritmo para la notación que utilizamos será de **O(n!).**

Estudio Experimental

TSPBacktrackingTodos O(n!)

n (nº vértices)	$\bar{T}(n)$ (ns)
14	777165
16	2816165
18	7706505
20	23548535
22	82761065
24	245016095
26	955016890
28	3422589130

Tiempo de Ejecución VS Tamaño de la Entrada

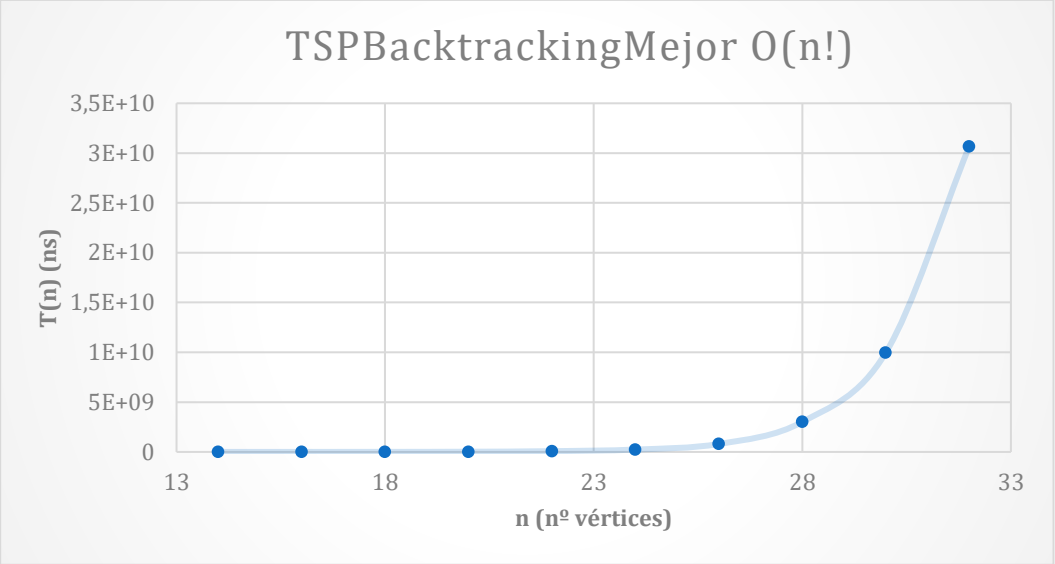


Gráfica tiempo de ejecución VS Tamaño de la entrada

TSPBacktrackingMejor O(n!)

n (nº vértices)	$\bar{T}(n)$ (ns)
14	587930
16	2407145
18	6267825
20	19684380
22	76872630
24	236059690
26	799495020
28	3017062855

Tiempo de Ejecución VS Tamaño de la Entrada

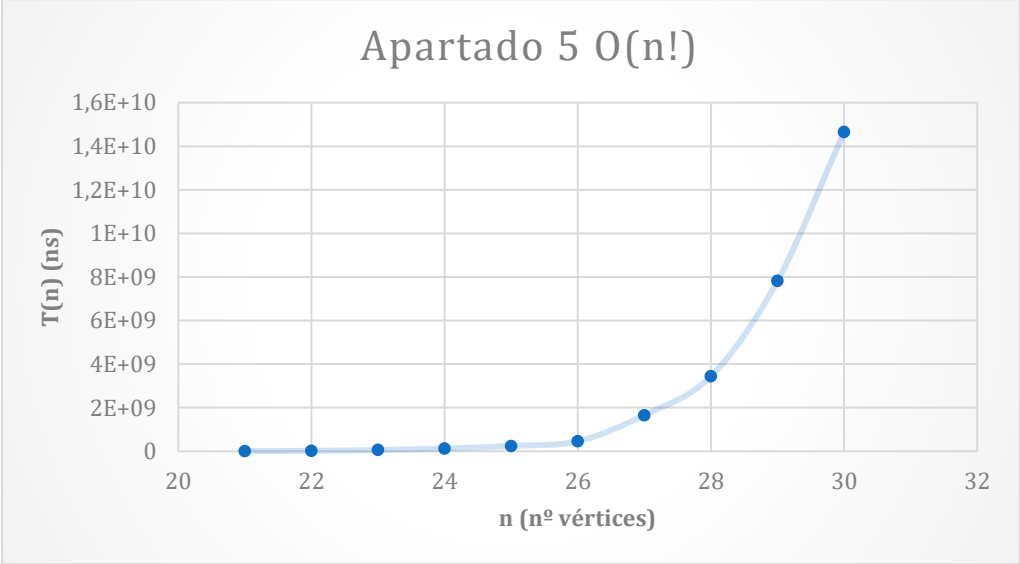


Gráfica tiempo de ejecución VS Tamaño de la entrada

Apartado 5 O(n!)

n (nº vértices)	T̄(n) (ns)
21	7763375
22	19417475
23	59342840
24	129099855
25	242923870
26	460858410
27	1659894235
28	3446416100

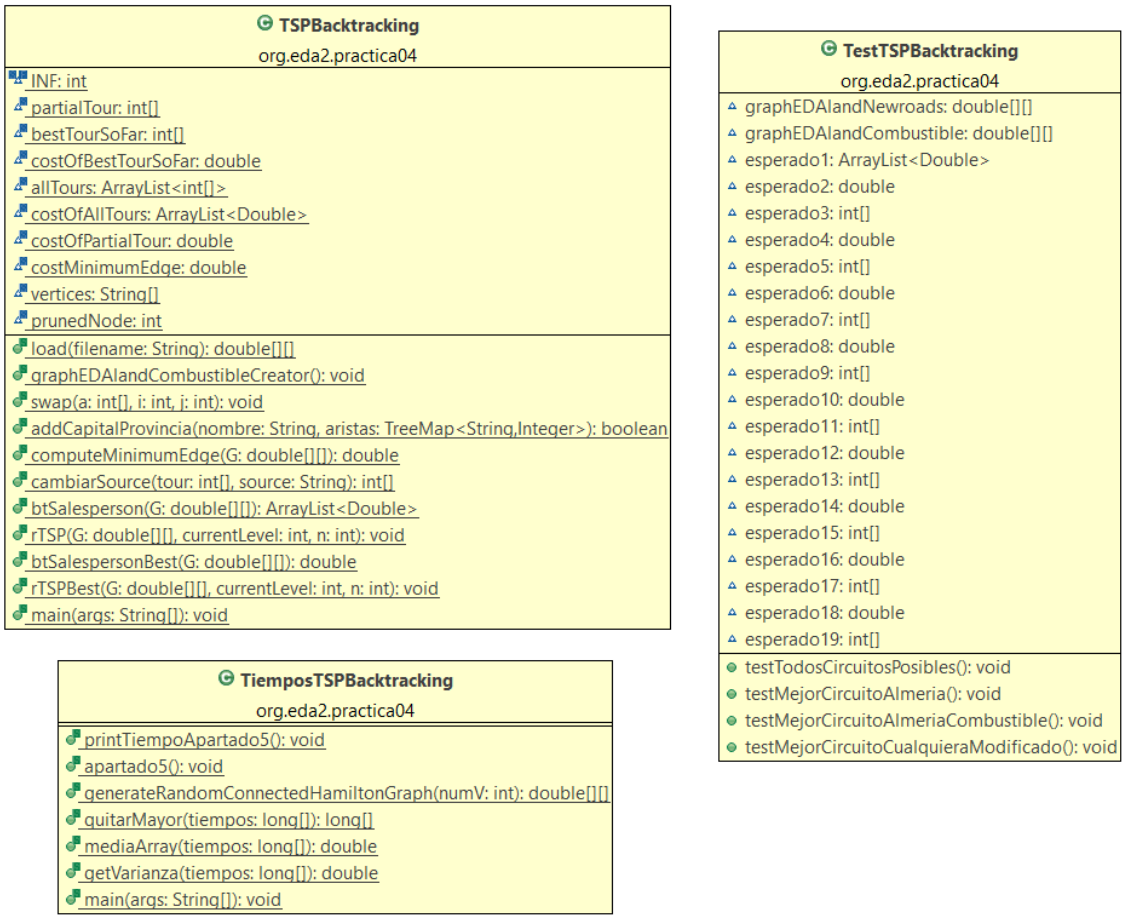
Tiempo de Ejecución VS Tamaño de la Entrada



Gráfica tiempo de ejecución VS Tamaño de la entrada

Anexo

Diagrama de clases



Archivos fuente:

- **TSPBackTracking:**

En este archivo dispuesto como clase java podremos encontrar los 3 métodos exigidos durante la práctica destinados a la orientación y gestión de los problemas propustos, junto a una serie de métodos (load, swap…).

- **TiemposTSPBackTracking:**

En este archivo dispuesto como clase java podemos encontrar una amigable interfaz que nos permite comprobar el tiempo de cualquiera de los algoritmos de la práctica con el n (número de vértices del grafo que queremos probar) que deseemos agregar.

- **TestTSPBackTracking:**

En este archivo dispuesto como clase java observamos la existencia de una serie de diversos test Junit 5 que nos permiten probar nuestro código con una variedad de nodos generados de forma aleatoria cuyo valor escala hasta 42 para probar así nuestro código de la forma más completa posible.

Fuentes Bibliográficas

1. Skiena, S. S. (2008). The Algorithm Design Manual. Springer Science & Business Media.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
3. LaMarca, A., & Robson, D. (2012). Backtracking Algorithms. Encyclopedia of Algorithms. Springer.
4. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
5. Beigel, R., & Eppstein, D. (1999). NP-completeness. A Guide to the Theory of NP-Completeness, 45-114.