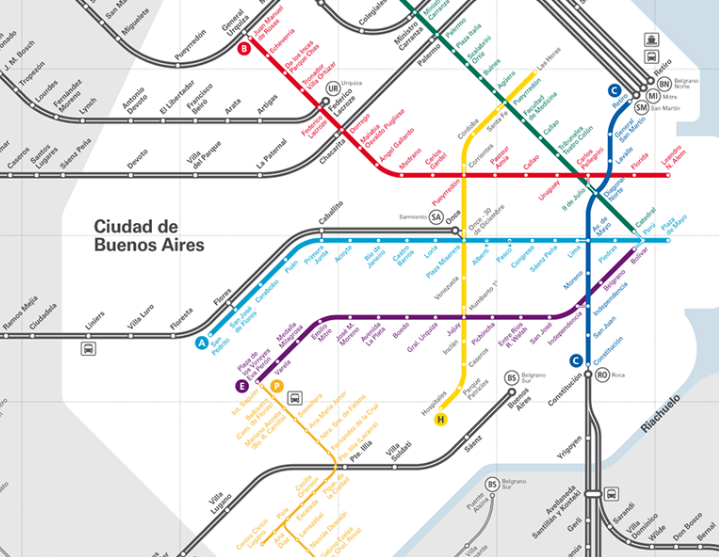
Trabajo práctico programación 3

Greedy vs Backtracking



Unicen TUDAI

Grupo Nro: 8:

Participante: Renzo Guerra Lattour

Backtracking:

**Estrategia utilizada:**

Se utilizó de forma auxiliar una estructura tipo “Pila”, donde en una de estas pilas guardábamos “la mejor solución hasta el momento” y en otra se guardaba la “solución actual” (la que se iba confeccionando mediante la recursividad).

En cada iteración se debía comprobar si la “solución actual” era o no mejor que la “mejor solución hasta el momento”. Debido a que al principio no existe una mejor solución, en el momento de verificar si “solución actual” tenia menor sumatoria de longitud de caminos que “mejor solución” debíamos de verificar si efectivamente ya se había encontrado anteriormente una **solución viable**, con esto nos referimos a que previamente se haya encontrado un conjunto de caminos el cual una todas estaciones, a su vez, se sabia que la cantidad de caminos debía ser de n-1, siendo n la cantidad de estaciones a unir. Por último paso, se debía verificar si, una vez elegidos los n-1 caminos candidatos (siendo n la cantidad de estaciones), realizando la estrategia unionFind ninguno de estos n-1 caminos eran redundantes (con redundantes me refiero a que no se contruyan caminos los cuales unan ciudades que ya a través de los caminos previamente elegidos se puede llegar desde A a B), como a su vez, que las n estaciones queden en un solo conjunto (dentro del método que valida la solución actual, se crea una instancia de unionFind cargando las n estaciones, las cuales comienzan todas separadas).

Se confeccionó un par de podas:

1- Validar que la cantidad de caminos en “solución actual” debe ser menor a (cantidad estaciones - 1).

2- La sumatoria de los caminos(km) de “solución actual” sea menor que la sumatoria de caminos(km) de “mejor solución” (solo en el caso que ya se haya encontrado una solución viable, y se haya almacenado esa solución en “mejor solución”).

3- Que la suma de cantidad de caminos seleccionados en “solución actual” con la cantidad de “caminos a elegir” sea mayor o igual a la cantidad de estaciones – 1 (con esto garantizamos no explorar “soluciones actuales” que como máximo lleven a un total de seleccionar n-2 caminos (los cuales son insuficientes)).

**Costo computacional:**

Es parecido al problema “pares de la suma de conjuntos” (PPS), donde de los n caminos dados en el txt, se puede tomar 2 acciones por cada camino, tomarlo o no tomarlo, por lo tanto, el costo computacional seria de:

Donde n es la cantidad de caminos dados en él .txt

(Claro está, que tras podas el costo computacional desciende).

Greedy:

**Estrategia utilizada:**

La idea fue que dados los ‘n’ caminos del txt, almacenar y ordenar los caminos de menor a mayor (teniendo como criterio la distancia) para luego instanciar un unionFind con las ‘j’ estaciones.

Se iba quitando 1 a 1 los elementos que se organizaron ascentendemente (el de menor valor primero) y verificar si ya había forma de llegar desde la estación A a la estación B, en caso de que no, se agregaba ese camino a una lista y se “unian” el conjunto que contenía la estación A con el conjunto que contenía la estación B (internamente terminan teniendo la misma raiz), en caso de que las 2 estaciones pertenecían al mismo conjunto, se descartaba el camino actual y se seguía con el siguiente, así hasta que los n caminos pertenecieran a un mismo conjunto.

Resulta que esa solución ya existe y se llama algoritmo de Kruskal, el cual, según Wikipedia dice que “*El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo*”.

**Costo computacional:**

Debido a que se toma el costo computacional mas alto, no importa que el mergeSort tenga una complejidad de n\*log en base 2 de n siendo n la cantidad de caminos, ni que el uniónComplete también tenga un costo computacional de n, ya que no se suman todos los O() internos, solo el mayor, por lo tanto el costo computacional es:

Donde n es la cantidad de caminos dados en él .txt

**Métricas:**

La métrica que se utilizó es la de cantidad de veces que itera cada uno de los métodos, en el Greedy la cantidad de veces que itera en el while, mientras que en el Backtracking es la cantidad de veces que se llama recursivamente el método privado auxiliar.

Con el data set 1:

Texto

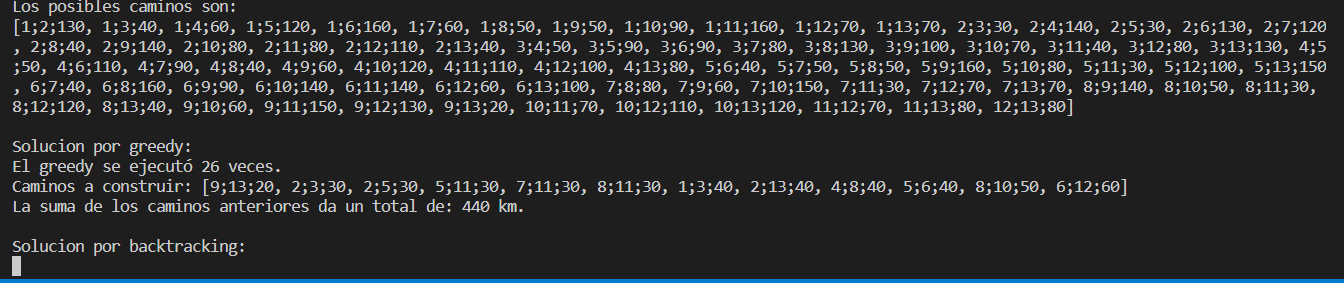
Descripción generada automáticamente

Con el data set 2:

Texto

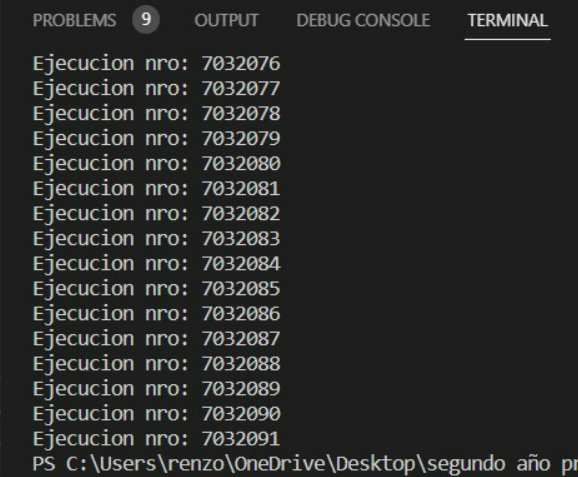
Descripción generada automáticamente

Con el data set 3:



El backtracking nunca termina de ejecutar, dado que su complejidad es de: .

Dejo abajo una prueba de que no termina más…



Se cortó la ejecución manualmente.

**Conclusión:**

Estuvo bueno el trabajo, ya que en este pude aprender a implementar tanto el “UnionFind” como el algoritmo de Kruskal. Se demostró que a pesar de que un problema se puede resolver de mas de una sola forma, en términos de eficiencia, es mucho más recomendable encarar ciertas resoluciones por sobre otras. Como también aprendí que plantear soluciones con pseudocódigo es mucho más fácil que pasar ese pseudocódigo a la computadora para que realmente funcione.