TP3 - Algo III - Mejorando el tráfico

Martín Acuña - Cristian Antonio - Manuel Horn

Junio 2023

1 Presentación y descripción del problema

Se una ciudad D con calles unidireccional (no hay puntos conectados en ida y vuelta) con "n" puntos y "m" calles que los unen con sus respectivas longitudes , dados además "k" calles con longitud candidatas que involucran puntos existentes en D, estas últimas de ida y vuelta entre los puntos involucrados (si ya existía en la ciudad original la ida o la vuelta entre esos 2 puntos solo considero la que falta, de no existir considero 2 calles de ida y de vuelta entre los puntos). Dados además dos puntos especiales "s" y "t", se nos pide devolver la distancia mas corta entre "s" y "t" entre los caminos resultantes de agregar cada calle (o par) candidata a la ciudad original, es decir $min\{d_D(s,t), d_{D+k_1}(s,t), ..., d_{D+k_k}(s,t)\}$ o -1 de no existir camino entre "s" y "t".

2 Descripción del algoritmo

Modelamos el problema con un digrafo de "n" vértices (puntos) y "m" aristas (calles) con peso (longitud entre los puntos de la arista). Utilizamos el algoritmo de Dijkstra para nuestra solución el cuál modificamos para que devuelva explícitamente el vector de distancias al vértice inicial, luego inicializamos el digrafo, su inverso y el vector de candidatas. Corremos Dijkstra desde "s"(en el grafo inicial) y desde "t" (en el grafo inverso) y guardamos las distancias en dos vectores. Por último recorremos el vector de candidatas y comparamos la distancia mínima actual (inicializado en la distancia de "s" a "t" en el grafo inicial) con las suma $d(s,v)+c(v->w)+d(t,w)$ y $d(s,w)+c(w->c)+d(t,v)$ para cada arista (v,w) candidata, en caso de encontrar un camino mas corto actualizamos la variable, de no existir camino entre "s" y "t" devolvemos -1. Notar que de esta forma solo corremos Dijkstra dos veces (accediendo a cada distancia en $O(1)$) y no cada vez que queremos calcular el camino mínimo para cada grafo modificado.

3 Justificación de correctitud

Dado un digrafo D, para todo "s" y "t" par de vértices en D, el camino mínimo es aquel que minimiza la suma de las longitudes de las aristas que lo componen, este lo obtenemos con Dijkstra. Para ver si las calles nuevas a utilizar mejoran mi distancia inicial solo necesito chequear las desigualdades $d(s,v) + c(v- > w) + d(t,w)$ y $d(s,w) + c(w- > c) + d(t,v)$ para cada calle (v,w) candidata . Esto es porque el mejor camino entre "s" y "t" que pasa obligatoriamente por (v,w) es aquel que minimiza las distancias entre "s" y "v", y "t" y "w", las cuales ya fueron calculadas con Dijkstra en el grafo original, esto vale porque solo podemos pasar por una candidata.

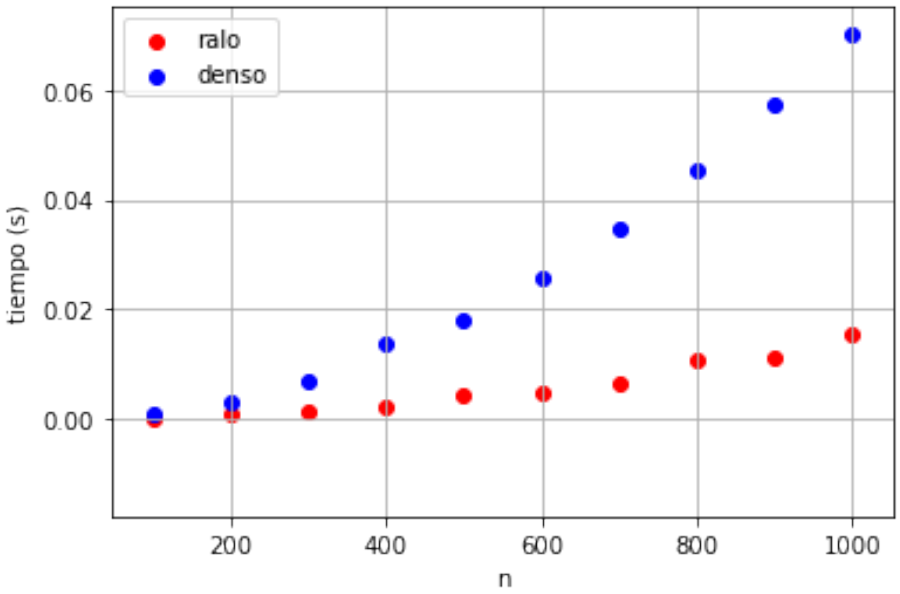
4 Análisis teórico y empírico de la complejidad

Implementamos el algoritmo de Dijkstra con tres estructuras distintas. Una matriz de adyacencias (n2), una cola de prioridad (pq) representada con un min-heap y un conjunto (set) representado con un red-black tree.

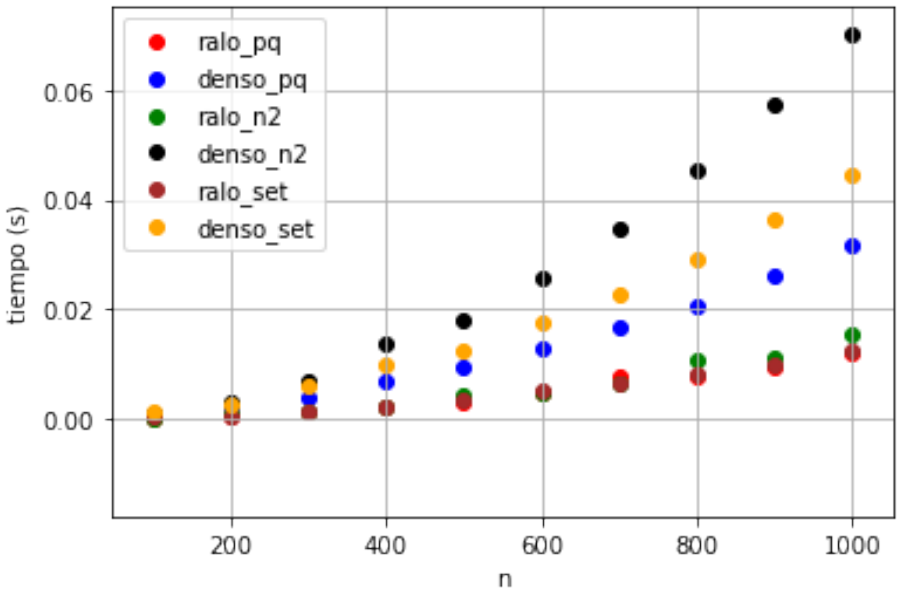
Teóricamente, la implementación con la matriz tiene complejidad temporal $O(n^2)$ mientras que las implementaciones set y pq tienen una complejidad $O(mlog(n))$, con m las aristas del grafo y n los vértices.

El análisis que llevamos a cabo consiste en comparar instancias de grafos densos contra grafos ralos. Es decir, para los grafos densos tenemos que $m = O(n^2)$ y para los ralos, $m = O(n)$.

Así, incrementamos la cantidad de vértices desde 100 hasta 1000, y así comparamos la complejidad experimental de los algoritmos para poder comparar esto con la teoría. A continuación presentamos como quedaron los tiempos de ejecución en función del tamaño de entrada.



a chequear qué es este primer gráfico (yo lo volaría)



Lo que podemos observar es que para las instancias de grafos ralos, las tres implementaciones en la práctica tienen tiempos parecidos. Sin embargo, para los grafos densos, la diferencia entre los tiempos de ejecución está más marcada.

Algo interesante que podemos observar es que, si bien las complejidades teóricas de pq y set son la misma, en la práctica la cola de prioridad es mejor porque es una estructura más eficiente (constantes que se ignoran en la complejidad teórica).

5 Conclusiones

La mejor implementación que usamos para la experimentación en términos de complejidad temporal es la de la cola de prioridad. Sin embargo, esto se puede mejorar un poco más cambiando la estructura min-heap por un Fibonacci min-heap. Esta estructura hace que la complejidad final sea $O(nlog(n))$, contra la del min-heap, que es $O(mlog(n))$. Así, para grafos densos, Fib-heap se mantiene en $O(nlog(n))$, mientras que en el min-heap tenemos $O(n^2log(n))$.