DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Informe Trabajo Practico 3: "Algoritmos sobre Camino Mínimo"

Análisis del Algoritmo de Camino Mínimo propuesto

17 de Junio de 2023 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Jarabroviski, Tomas	325/21	jarabroviskit@gmail.com
Cravchik, Matias	720/21	cravchikm@gmail.com

Abstract

Presentaremos y haremos una descripción del problema planteado para luego explicar el algoritmo desarrollado, justificando la correctitud del mismo mediante el invariante del algoritmo seleccionado. Para luego experimentar en base a distitnas implementaciones del algoritmo, dependiendo de su caso de uso y complejidad.

Keywords: Algortimos, Estructuras de Datos, Camino Mínimo, Dijkstra, Recorridos, Pruebas empíricas, Experimentación, Complejidad, Técnicas Algorítmicas.

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Introducción

El problema a resolver es el siguiente: tenemos n puntos de una ciudad (que podrían representar esquinas) unidos por m calles unidireccionales que conectan 2 de ellos. Nos dán, además k calles bidireccionales que se proponen construir para minimizar la distancia entr dos puntos críticos s y t. Tenemos que hacer un algoritmo que, dados $n \leq 10^4$, $m \leq 10^5$, $k \leq 300$, $1 \leq s \leq n$, $1 \leq t \leq n$, las calles unidireccionales con su longitud y las calles bidireccionales propuestas (también con su longitud) nos diga la mínima distancia de s a t sabiendo que se cuenta con la posibilidad de construir una de las calles propuestas, o -1 en caso de que no haya un camino de s a t.

Notar que entendemos que las calles unidireccionales son estrictamente unidireccionales por lo que si se incluye la calle $(u \to v)$ no estaría la calle $(v \to u)$. Esto también limita la cantidad de máxima cantidad de calles a $(n * (n-1))/2$.

2. Explicación del algortimo desarrollado

2.1. Algoritmo

Para resolver el problema nos construimos un dígrafo en donde cada nodo representa uno de los n puntos. Luego cada calle unidireccional la representamos con una arista de un nodo a otro (con el sentido y longitud correspondiente), de esta forma modelando directamente las calles de la ciudad.

Una vez que tenemos este dígrafo calculamos el camino mínimo desde s al resto de esquinas (con el algoritmo de Dijkstra, por ejemplo) guardandonos las distancias a s en un vector. De igual forma, calculamos el camino mínimo desde t al resto pero en el digrafo con el sentido de los arcos invertidos, efectivamente obteniendo la distancia de todas las esquinas hasta t.

Una vez que tenemos esto, vamos iterando por las calles bidireccionales nuevas y ver cual de ellas minimíza más la distancia de s a t (en caso que alguna lo haga). Es decir, nos quedamos con la mínima distancia de probar entre todas las aristas bidireccionales: $(u \to v)$ tal que la $d(s \to u) + c(u \to v) + d(v \to t)$ ó $d(s \to v) + c(v \to u) + d(u \to t)$ (pues es bidireccional) sea la menor. En caso que ninguna arista bidireccional mejorase la longitud del camino mínimo respecto del original, devolvemos la longitud del camino mínimo original.

Notar que si no existiese camino de algun nodo a otro la distancia entre esos dos nodos sería infinita. Por lo cual, si no existiese camino de s a t aún con alguna calle bidireccional agregada, la distancia sería infinita y devolvemos -1.

3. Justificación de Correctitud

Al correr un algoritmo de camino mńimo sobre s tendremos la distancia del camino mínimo de s al resto de las esquinas, incluyendo t. Además, al hacer lo mismo sobre t con las direcciones de las calles invertidas sabremos la longitud del camino mínimo de todas las esquinas a t (pues la distancia de un camino en una dirección es igual a la distancia del camino en la dirección opuesta). Para calcular los caminos mínimos podemos usar Dijkstra ya que todas las calles tienen longitud positiva.

Luego al recorrer cada calle bidireccional propuesta, las podemos entender como 2 calles unidireccionales con la misma longitud: cada calle bidireccional uv se puede ver como una calle $(u \to v)$ y otra $(v \to u)$. Además sabemos que nunca tomaremos ambas ya que terminariamos aumentando la longitud del camino sin cambiar de esquina, pues $d(s \to u) + c(u \to v) * 2 + d(u \to t) > d(s \to u) + d(u \to t)$. Por lo tanto si una de las calles mejora la longitud del camino mńimo significa que se el camino mńimo de s a t con la calle bidireccional uv pasa por $(u \to v)$ o por $(v \to u)$ (sino no habría diferencia con incluirla o no); si pasa por $(u \to v)$, por ejemplo, entonces el camino minimo de s a t será un camino minimo de s a u seguido de $(u \to v)$ y luego un camino minimo de v a t, con una longitud total de $d(s \to u) + c(u \to v) + d(v \to t)$. Esta debe ser la longitud del camino mńimo inclutendo la arista bidireccional ya que cualquier otro camino C de s a t que pase por $(u \to v)$ tendra una longitud de $d_C(s \to u) + c(u \to v) + d_C(v \to t)$ (d_C las distancias en C) con $d_C(s \to u) \geq d(s \to u)$ y $d_C(v \to u) \geq d(v \to t)$ y por lo tanto no puede ser más corto.

El algoritmo es correcto ya que luego de iterar arista por arista bidireccional en busca de un camino de menor longitud siguiendo las calles en alguno de sus dos sentidos estamos seguros de que nos quedaremos con el menor de todos, pues que probamos con todas las posibles nuevas calles.

4. Experimentación

4.1. Análisis teórico de las implementaciones de camino minimo

Conocemos dos algoritmos que nos permiten calcular el camino mínimo de un nodo fuente a todo el resto: los algoritmos de Dijkstra y Bellman-Ford. El algoritmo de Dijkstra cuenta con diferentes implementaciones con distintas complejidades teóricas, la versión sin cola de prioridad tiene una complejidad de $O(N^2)$, mientras que al incluirla esta puede cambiar dependiendo de cómo esté implementada la misma; utilizando un árbol binario la complejidad será de $O((M+N)logN)$ y utilizando un Fibbonacci-Heap será de $O(NlogN+M)$. El algoritmo de Belman-Ford, por otro lado, siempre tendrá una complejidad de peor caso de $O(M*N)$, aunque puede tener optimizaciones que mejoren su tiempo en la práctica, por ejemplo contando con la capacidad de parar cuando no se realize ninguna relajación en una iteración (no se actualize el vector de distancias).

Esto nos dice que cabría esperar que el algoritmo de Dijkstra con cola de prioridad (en ambas versiones) se ejecute más rápido que sin ella y que Belman-Ford cuando el grafo sea relativamente ralo, es decir, halla pocas calles en comparación a la cantidad de esquinas. De igual forma, estas deberían ser superadas por la versión sin cola de proprida cuando el grafo sea muy denso ($M\ O(N^2)$), siendo seguida por la implementación con Fibbonacci-Heap. Belman-Ford, en contraste, cuenta siempre con la peor complejidad, aunque al ser capaz de parar antes podría llegar a comportarse de manera similar.

Cabe destacar, además, que todas las implementaciones requieren primero del armado del grafo con los datos de entrada, cuya complejidad dependerá de la estructura utilizada para el modelado.

4.2. Análisis empírico de los algoritmos

Para este análisis medimos el tiempo que toma generar el modelo del grafo y correr el algoritmo de camino minimo (tanto sobre el nodo de salida como el de llegada con las calles invertidas), tomando el promedio de diez corridas para los distintos algoritmos mencionados: sin cola de prioridad (utilizando una matriz de adyacencia), con priority-queue (usando listas de adyacencia) de la libreria estandard de C++, implementada con un árbol binario, con un Fibbonacci-Heap (también con listas de adyacencia) de https://github.com/robinmessage/fibonacci y Bellman-Ford (con lista de aristas) parando si no hay relajación.

Para ver las diferencias entre los algoritmos generamos tests con 1000 nodos y calles aleatorias para distintas densidades (cantidad de calles / cantidad máxima de calles), incluyendo al menos un camino de s a t. Estos fueron los resultados:

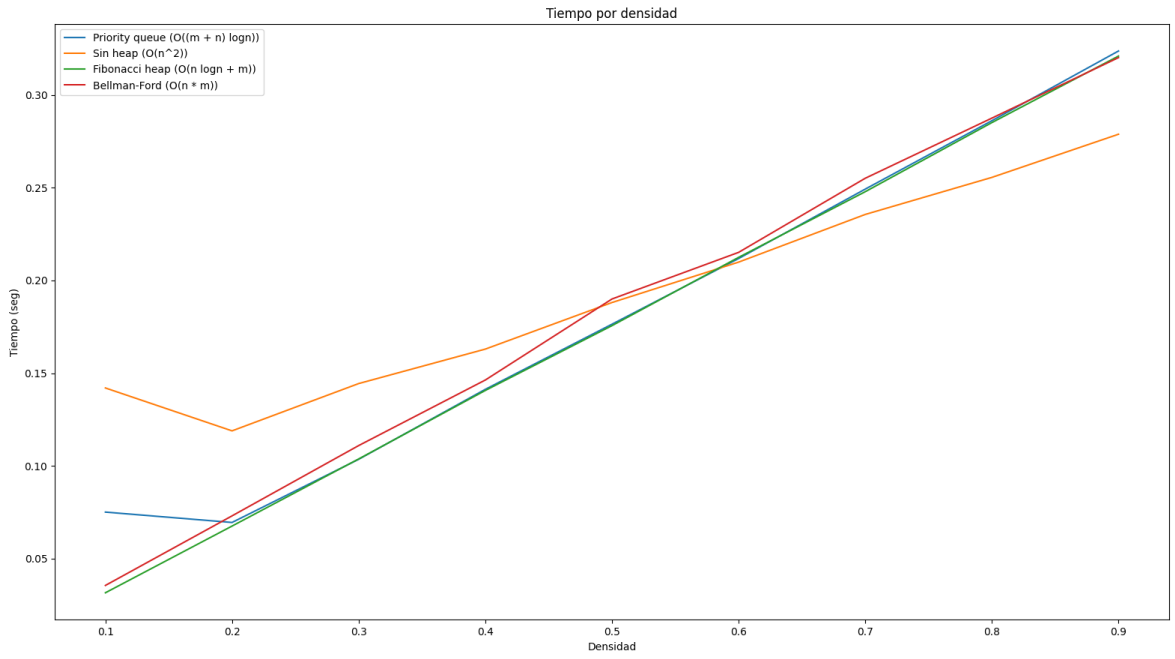

Figura 1: Tiempo de ejecución (en segundos) por densidad para distintas implementaciones de camino mńimo (1000 esquinas)

Como se puede observar, la versión de Dijkstra sin cola de prioridad fue notablemente más lenta para densidades menores a 0,5 pero superó al resto para las densidades mayores a 0,6. Bellman-Ford se acercó en momentos bastante al resto de implementaciones de Dijkstra, siendo sin embargo ligeramente más lento y un poco más inconsistente. Ambas versiones de Dijkstra con cola de prioridad dieron resultados muy similares, notando una ventaja del Fibbonacci-Heap en la menor densidad y otra casi imperseptible en la mayor.

Si bien esto nos dá una idea de como se comportan los algoritmos para estas cantidades de esquinas, hay que notar que si bien la especificación del problema permite hasta 10000 esquinas, esta limita hasta 100000 calles por lo que nunca se llegaría a las densidades más altas en un test con tantas esquinas. Por esto medimos también lo que ocurre con 448 esquinas, la máxima cantidad que permite llegar a una densidad de 1.

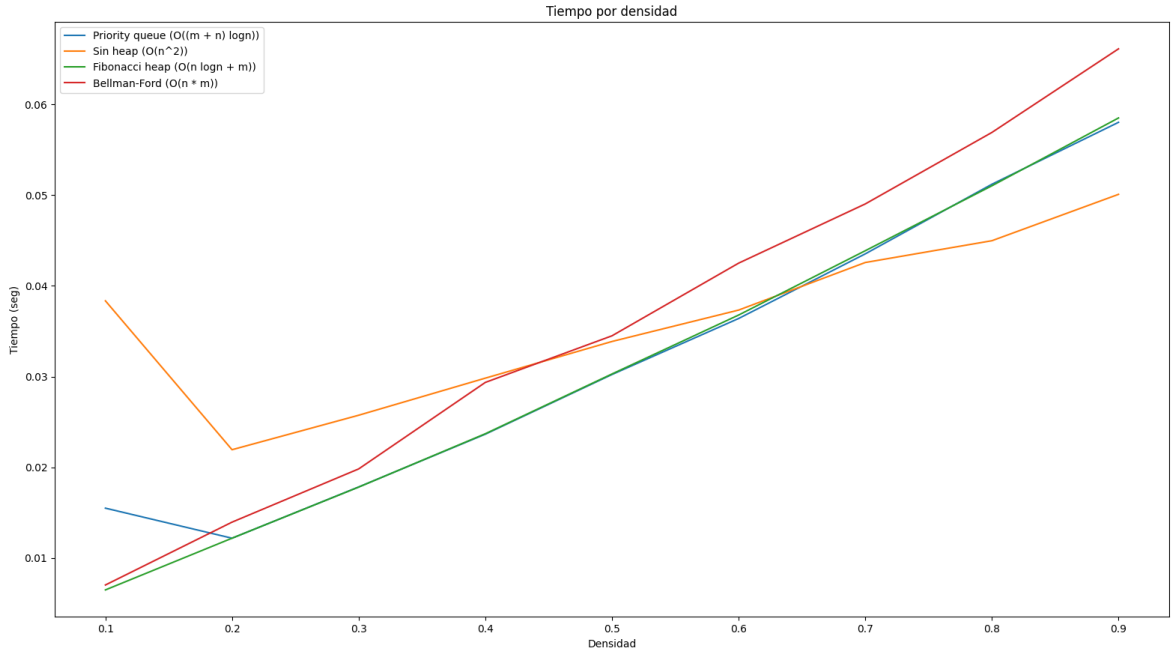

Figura 2: Tiempo de ejecución (en segundos) por densidad para distintas implementaciones de camino mńimo (448 esquinas)

Los resultados en este caso son muy similares, destacando una mayor diferencia entre Bellman-Ford y el resto. La diferencia más importante, sin embargo, es que toma ahora una densidad de 0,7 para que Dijkstra sin cola de prioridad supere al resto.

Otro caso que puede ser interesante es lo que ocurre con una baja cantidad de nodos, por lo que realizamos algunas mediciones con 100 esquinas:

Densidad	Priority-Queue	Sin queue	Fibbonacci-Heap	Bellman-Ford
0.1	0.000907581	0.00221220	0.000862581	0.000804365
0.9	0.00603033	0.00565405	0.00582545	0.00652658

En este caso Belman-Ford es el superior cuando hay poca densidad, proablemente debido a que puede terminar con pocas iteraciones. En densidad alta, Dijkstra con Fibbonacci-Heap se acerca bastante a la version sin cola, explicable por su complejidad ya que $O(NlogN + M)$ tenderá a $O(N^2)$ cuando $M \in O(N^2)$.

Como conclusón, las versiones de Dijkstra con cola de prioridad parecieran ser las más recomendables para la mayoría de casos esperables, salvo para los menos frecuentes casos de muy alta densidad. Entre estas, la versión con Fibbonacci-Heap puede ser la mejor elección ya que su complejidad teórica indica que se escalará mejor con altas densidades y cantidad de nodos.