DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Camino Mínimo y Flujo Máximo

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

Integrante	LU	Correo electrónico
Teplizky, Gonzalo Hernán	201/20	gonza.tepl@gmail.com

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

Resumen

Para el tercer y último trabajo práctico de la materia, el enfoque será puesto en trabajar con nuevos algoritmos de Grafos. Entre ellos, se encuentra el famoso problema de camino mínimo sobre digrafos pesados entre un vértice y los demás. También trabajamos sobre redes para calcular un Flujo Máximo en la misma, al que utilizaremos como base para modelar otros problemas cuya resolución es factible mediante este, con implementaciones vistas en la materia.
En cuanto a los ejercicios de este trabajo, el desempeño será testeado por un *juez online*[1].
Este informe se centrará en el primer ejercicio del trabajo práctico, donde se nos pide modelar un problema que se resuelve con Camino Mínimo, implementado con distintintas estructuras de datos y algoritmos. La estructura es la siguiente:

- **Introducción - Presentación del problema**: Se da un vistazo inicial del problema a resolver.
- **Desarrollo - Algoritmo. Correctitud y Complejidad del mismo**: Se explica el algoritmo desarrollado para la resolución del problema, justificando la complejidad computacional y la correctitud del mismo.
- **Experimentación y resultados**: Se compara empíricamente el funcionamiento del algoritmo para distintas implementaciones y estructuras de datos que permiten resolver el problema de camino mínimo.

Palabras clave: *Grafos*, *Recorridos*, *Camino Mínimo*, *Dijkstra*, *Bellman-Ford*, *Flujo Máximo*, *Corte Mínimo*, *Ford-Fulkerson*, *Edmonds-Karp*.

Índice

[1] `https://www.spoj.com`

1. Ejercicio 1

1.1. Introducción - Presentación del problema

En este ejercicio se nos presenta el mapa de una ciudad, representada por un conjunto de N puntos numerados, del 1 al N. Para unir estos puntos, contamos con M calles unidireccionales, cada una con cierta longitud, que conectan, cada una, uno de estos puntos, con otro.
Dados dos puntos críticos de la ciudad, S y T, se pretende reducir la longitud del camino total que sale de S y llega a T. Y para esto se propone la construcción de una nueva calle, bidireccional, en la ciudad. Esto quiere decir que por cada calle (v, w) candidata a mejorar el camino entre s y t, con longitud l, tendremos dos calles unidireccionales: $v \to w$ y $w \to v$, cada una de longitud l.
Para esta nueva calle tenemos K alternativas posibles, cada una con su longitud corrrespondiente entre los dos puntos de la ciudad que conecta.
Se pretende decidir, entre esas K posibles construcciones, cuál es la que mayor reducción del camino entre S y T produce. En concreto, se desea obtener el nuevo valor del camino entre estos dos puntos:

- En caso de que alguna de las K opciones efectivamente acorte el camino entre S y T, queremos devolver la longitud del camino total resultante, luego de construir la nueva calle que minimice, de entre todas las K opciones, esta longitud.
- Si la construcción no permite reducir la longitud del camino entre S y T, se mantiene la longitud del camino mínimo que poseíamos previo a considerar las calles bidirecionales. Si directamente no existía camino entre S y T, y la adición de las bidireccionales tampoco nos lo dio, esto se indica devolviendo -1.

1.2. Algoritmo

Para resolver este problema, modelamos la ciudad mediante un grafo dirigido pesado, D:

- $V(D)$ es el conjunto de los N puntos de la ciudad. Notar que $S \in V(D)$ y $T \in V(D)$.
- $E(D) =$ es el conjunto de las M calles unidireccionales que tiene la ciudad. El costo de cada arista (v, w) es la longitud de la calle que conecta el par de puntos v con w $(v, w \in V(D))$.

Por otro lado, las nuevas calles que son candidatas a mejorar el camino mínimo entre S y T, lo representamos mediante un conjunto de aristas pesadas C, con $|C| = K$
Para poder analizar qué arista de este conjunto mejora el camino mínimo entre los puntos críticos de la ciudad S y T, debemos primero averiguar la distancia entre este par de vértices. Esto podemos hacerlo mediante alguno de los algoritmos de camino mínimo conocidos, entre un vértice del digrafo y todos los demás.
Como las longitudes de las calles son siempre positivas, sabemos que los costos de las aristas de D siempre serán positivas. Por lo tanto, podemos utilizar el algoritmo de $Dijkstra$ para hallar el camino mínimo entre un vértice y todos los demás de un (di)grafo pesado. Equivalentemente, dado un vértice y el (di)grafo traspuesto del original, nos devuelve el camino desde todos los vértices del (di)grafo hacia ese.
Una vez que ya tenemos las distancias entre cada par de vértices y S y T (luego de dos corridas del algoritmo de Dijkstra), podemos comenzar a evaluar las K propuestas de construcción de nuevas calles bidireccionales en la ciudad.
Llamamos d a la distancia mínima entre S y T encontrada hasta el momento. Antes de considerar nuevos accesos entre los puntos de la ciudad, tenemos que $d = d(S, T)$. Sea $e = (v, w)$ una candidata a minimizar la distancia entre estos puntos críticos, y $c : E(D) \to \mathbb{N}$ la función de costos asociados a las conexiones entre ellos. Si se cumple que

$$d(S, v) + c(v, w) + d(w, T) + < d$$

entonces e mejora el camino entre S y T. Ahora $d = d(S, v) + c(v, w) + d(w, T)$. Un pseudocódigo posible que ilustra esta idea, es el siguiente:

Notemos que si d es igual a la distancia que había originalmente entre S y T, esto implica

Algorithm 1 mejoraEnTrafico(D, C):

```
dS ← Dijkstra(D, S) //dS es el vector distancia de D partiendo desde S
dT ← Dijkstra(D, T) //dT es el vector distancia de D partiendo desde T
d ← dS(T) //d es la distancia mínima entre S y T encontrada hasta el momento
i ← 0
while i < K do
  //iteramos por cada arista C_i candidata a mejorar el camino entre S y T
  //para cada nueva calle bidireccional candidata (v,w) de longitud l, en C_i se encuentran dos
  calles unidireccionales, de longitud l cada una: v → w y w → v
  nuevaDistancia ← d(S, v) + c(C_i) + d(w, T)
  if (nuevaDistancia < d)
   d ← nuevaDistancia
   i ← i + 1
end while
return d
```

que ninguna calle candidata a ser construida, mejora el camino entre estos dos puntos críticos que tiene la ciudad

1.3. Correctitud del Algoritmo

Queremos devolver la nueva mínima longitud que puede tener un camino entre dos puntos críticos de la ciudad, S y T, si se construye alguna de las K calles bidireccionales candidatas.
En el modelo planteado, las distancias entre cada par de puntos de la ciudad están representadas por los costos de las aristas que inciden a ellos: si tengo una arista que conecta a u con v cuyo costo es w, entonces en la ciudad tengo que la calle que me permite llegar del punto u al punto v, tiene longitud w.
Por lo tanto, si quiero hallar la distancia mínima que hay entre un punto u y otro v cualquiera de la ciudad, esto es equivalente a buscar la distancia desde u hacia todos los demás puntos del grafo, entre los cuales se encontrará v. También vamos a necesitar la distancia desde todos los puntos del grafo, entre los cuales se va a encontrar u, hacia v.
De esta forma, tengo dos vectores de distancias de la ciudad, Du y Dv. El primero, dado un punto cualquiera de la ciudad x, me dice la distancia que me lleva ir desde el punto u hacia él. Por otro lado, Dv me indica la distancia que tengo hasta v, partiendo desde x.
A partir de esto, ya estoy en condiciones de poder evaluar las nuevas calles, candidatas a mejorar el camino mínima entre estos dos puntos críticos de la ciudad. Sea $c = (a, b)$ una de estas calles candidatas de la ciudad, c parte desde el punto a, y llega al punto b. Expresamos su longitud como $w = w(a, b)$.
Por cada calle bidireccional (a, b) candidata a mejorar el camino, consideramos dos calles unidireccionales $a \to b$ y $b \to a$, cada una de costo $c(a, b)$, es decir, la longitud de la calle bidireccional original.
La longitud del camino entre dos puntos u, v cualesquiera de la ciudad, puede reescribirse como $d(u, v) = d(u, x) + w(x, y) + d(y, v)$, siendo x, y dos puntos intermedios de este camino cualesquiera.
La calle c mejora la distancia entre u y v, si vale que la longitud del camino entre u y a, sumado a la longitud del camino entre b y v, sumado a w, es menor que $Du(v)$ (ó, equivalentemente, $Dv(u)$).
Luego, de todas aquellas candidatas que mejoran esta longitud total del nuevo camino entre u y v, nos quedamos con aquella candidata que minimice la nueva longitud.

1.4. Complejidades del Algoritmo

Para hallar el camino mínimo entre un vértice de D y los demás, como ocurre con el caso de S y T, utilizamos el algoritmo de Dijktra. La complejidad de este algoritmo, en peor caso, es de $O(min\{N^2, MLogN\})$. Por lo tanto, encontrar las distancias entre S y los demás vértices de D, y de T y los demás vértices de D, tiene este costo.
Luego, tenemos que recorrer las K aristas candidatas a mejorar la distancia entre los dos puntos críticos que tiene la ciudad. Por cada candidata, si consideramos la suma de las longitudes de las calles como constante en tiempo de ejecuciíon, tenemos que en $O(K)$ iteraciones, podemos chequear si alguna de las modificaciones propuestas mejora la longitud de la distancia entre los dos puntos de la ciudad S y T.
En nuestra implementación, la complejidad final del algoritmo es de $O(M * LogN)$.

1.5. Experimentación

En esta última parte del informe, nos propusimos llevar adelante la parte experimental de nuestro ejercicio, con el fin de probar la perfomance del mismo.
Si bien en la materia el algoritmo de Dijkstra es el que utilizamos siempre que queremos hallar camino mínimo desde algún nodo en un grafo o digrafo pesado cuando los pesos de cada eje son positivos dada su eficiencia, a la hora de pasar a la parte implementativa no solo que existen otros algoritmos que resuelven el mismo problema, si no que también hay diversas estructuras de datos para lograrlo.

En nuestro caso, para la entrega y el juez resolvimos el ejercicio implementando el algoritmo de Dijkstra de la forma en la que fue visto en la práctica, que es usando una *Cola de Prioridad* o *Priority Queue*, en particular, la versión mínima. Esta estructura nos permite extraer e insertar en $O(LogN)$ pares de tipo *(nodo, peso)*, con el fin de procesar en cada paso el nodo de menor peso actual entre los no procesados. Eso nos lleva a la complejidad que finalmente tiene el algoritmo, analizada anteriormente.
Para la experimentación, probamos tanto con un algoritmo de camino mínimo distinto, como con estructuras de datos diferentes bajo la misma implementación de Dijkstra. La idea es ver si la noción teórica acerca de las complejidades en cada caso, se cumple al analizar empíricamente con diversas instancias de test para los mismos.

Por eso, nos centraremos en:

- **Analizar teóricamente** que implementación y estructuras son mejores que otras según aspectos de eficiencia a nivel asintótico en los inputs.
- **Concluir empíricamente** como resultaron los tiempos de ejecución para distintas instancias probadas para las implementaciones y estructuras propuestas.

1.5.1. Análisis Teórico

Lo primero que pensamos fue en tomar la otra implementación en grafos pesados para camino mínimo de uno a todos, que es el algoritmo de **Bellman-Ford**. Como lo hemos visto en clase, este algoritmo nos provee una interfaz más robusta en grafos que pudieran tener aristas de costo negativo, detectando e informando cuando tenemos ciclos cuyo costo es negativo. Como en este ejercicio los pesos son longitudes de calles, el valor es siempre positivo, y eso nos permite usar Dijkstra, aprovechando su eficiencia. Dado que Dijkstra, como lo armamos, debe correr en $O(MLogN)$, y Bellman-Ford lo hace en $O(MN)$, creemos que Dijkstra irá más rápido sea denso o ralo el grafo.

Luego de cambiar la implementación, nos pareció interesante ver también la diferencia en la ejecución cuando, siempre dentro del algoritmo de Dijkstra, la estructura que maneja los nodos y sus pesos es modificada.

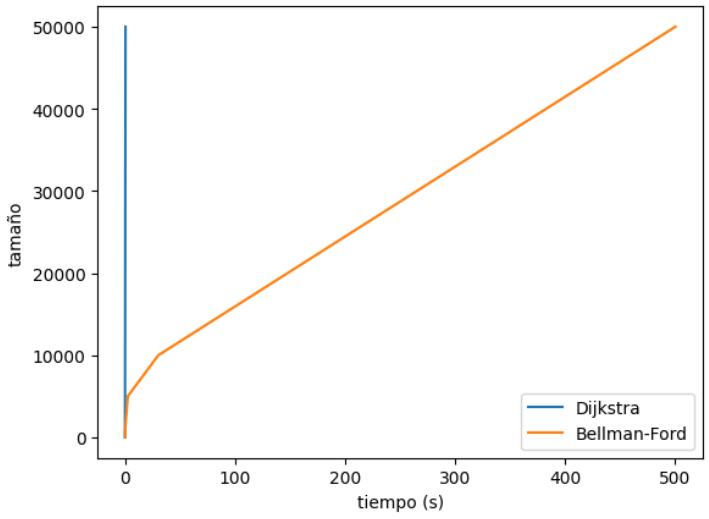
Buscamos una estructura que a nivel teórico, al menos, pudiera funcionar mejor que la cola de prioridad mínima que teníamos hasta ahora. El problema que tiene la estructura que usamos es que siempre que relajamos un nodo estamos insertándolo nuevamente en la cola con la nueva distancia, menor a la que tenía previamente, pudiendo tener varios duplicados por nodo dentro de la cola.
Si bien esto es a priori mejor que buscar en la cola al nodo y actualizar el valor, que es $O(n)$, cuando el grafo empiece a crecer va a ser más ineficiente lidiar con tanto duplicado.
Por eso propusimos la estructura de **cola de prioridad mínima indexada** en la cual solo tenemos una entrada por cada nodo. Esto busca evitar la presencia de duplicados, y nos permite agregar una entrada solo cuando no está, y actualizar el valor de un nodo existente, mejorando su camino mínimo, si no fue aún procesado, en $O(LogN)$. La teoría detrás de esto nos dice que la diferencia se tendría que notar cuando el grafo empiece a crecer considerablemente, no antes.

1.5.2. Análisis Empírico

Para testear los algoritmos y poder ver empíricamente si los tiempos se traducen en lo que teorizamos que debía suceder, nos armamos grafos más y más grandes. En particular, lo importante está en las ejecuciones para hallar camino mínimo con las aristas unidireccionales, ya que las bidireccionales son analizadas luego en un tiempo que mayormente no le va a ganar al de calcular camino mínimo, al menos en nuestra implementación. De modo que la importancia en el crecimiento del input radica en los nodos y aristas unidireccionales que se van agregando.

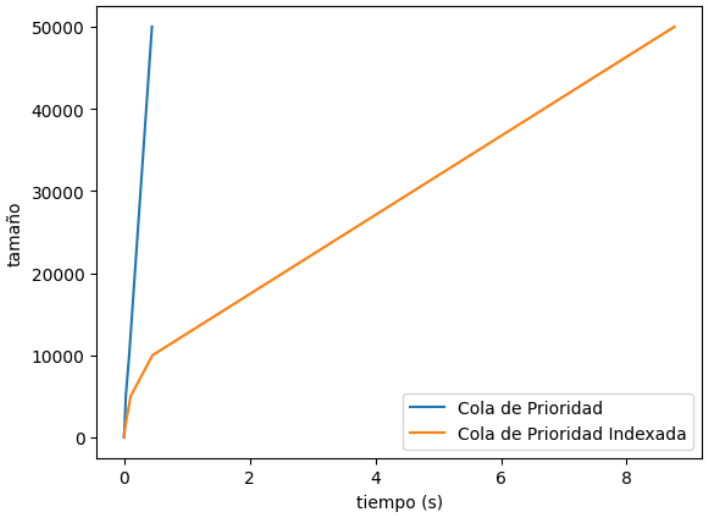
Para generar las instancias de test, utilizamos un script de Python que dejaremos anidado, el cual nos arma instancias cumpliendo las restricciones del enunciado, en donde se considera cantidad de nodos, y de calles unidireccionales y bidireccionales. Para los gráficos, nos basamos en la clase de experimentación de Greedies que hubo este cuatrimestre, con el agregado de dibujar más de una recta a la vez. Corrimos las distintas versiones con la libreria chrono para ir generando los segundos que tardan y luego graficar todo.

Comparemos por un lado los tiempos para **Dijkstra** y **Bellman-Ford** con un gráfico de tiempo-cantidad de nodos. Los tamaños de las instancias en cantidad de nodos son de n=10, n=100, n=500, n=1000, n=5000, n=10000 y n=50000.



Nos sucedió que muy rapidamente el algoritmo de Bellman-Ford empezó a converger, incluso con poca cantidad de nodos y aristas. Esto hizo que al llegar a cantidad de nodos como 10000 o más,

y multiplicado por 5 o 6 la cantidad de aristas, parezca que es casi constante Dijkstra contra un crecimiento mucho más pronunciado en Bellman-Ford. Sucedió lo que pensábamos pero de forma mucho más exagerada.
Veamos ahora como funciona el algoritmo de Dijkstra cuando utilizamos la **cola de prioridad mínima original** que armamos en el algoritmo entregado, y la estructura que agregamos para ir más rapido, que es la **cola pero indexada**, con un gráfico de igual característica que el anterior.



Si bien fue mucho más rapido que Bellman-Ford, la implementación de la cola de prioridad mínima indexada fue bastante peor que la original del algoritmo de Dijkstra, cuya recta dejo de verse constante pero no está muy lejos de serlo.
En conclusión, **la implementación vista en clase fue mucho mejor**, siendo que quizás esta estructura de cola de prioridad indexada, en otro lenguaje o con otro tipo de grafos, podría haber sido mejor, pero no en este caso.