DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP 3: Camino mínimo y Flujo máximo

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

Integrante	LU	Correo electrónico
Zaid, Pablo	869/21	pablozaid2002@gmail.com
Arienti, Federico	316/21	fa.arienti@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

En la teoría de grafos, el problema del *camino mínimo*[1] se refiere a una serie de problemas relacionados a encontrar, para un grafo —o digrafo— $G = (V,\ E)$ con función de peso $w : E \to \mathbb{R}$ asociada y ciertos pares de vértices $\mathcal{C}$, un conjunto de caminos $s \rightsquigarrow t$, $(s,\ t) \in \mathcal{C}$, para los cuales la suma total del peso de sus aristas —su *distancia*— es mínima de entre todos los caminos posibles con esos extremos. En este informe, nos vamos a concentrar en la variante del problema conocida como *camino mínimo con una única fuente*, donde interesa conocer la distancia de cualquier camino mínimo entre un vértice $s \in V$ y todo el resto de los vértices $w \in V \backslash \{s\}$.

Existen diversos métodos para la resolución de este problema. Entre ellos, los algoritmos *golosos* de *Bellman-Ford* y de *Dijkstra*, que se basan en el concepto de *relajación*[2] de aristas para la construcción de una solución.

El siguiente informe evalúa el problema del *tráfico*, explicado en el próximo apartado, y lo reformula como una aplicación particular del problema de *camino mínimo con una única fuente* que aprovecha la propiedad de subestructura óptima de los caminos mínimos. Además, evalúa la eficiencia de la solución propuesta de manera empírica.

Palabras clave: *camino mínimo, algoritmo de Dijkstra.*

Índice

[1] Ver Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest y Clifford Stein. Introduction to algorithms. 2009. Sección 24: *Single-source shortest paths.*

[2] Podemos pensar en el proceso de relajación como un método por el cual se mejora, sucesivamente, la cota superior de la distancia que puede tener un camino mínimo. El mismo se basa en la propiedad de desigualdad triangular: si $\delta : E \to \mathbb{R}$ denota la distancia mínima entre cualquier par de vértices en un grafo G, entonces para cualquier par de vértices s y t y arista $(u,\ t) \in E$ con $u \neq s$, $\delta(s,\ t) \leq \delta(s,\ u) + w(u,\ t)$.

1. El problema del tráfico

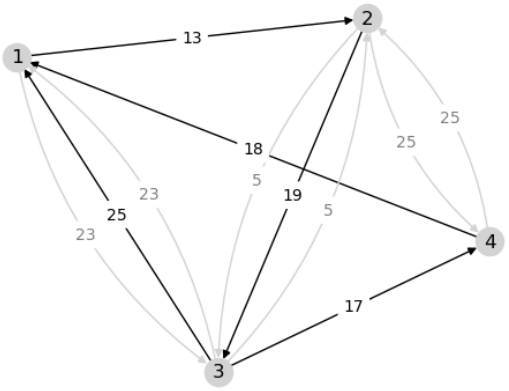
El problema del *tráfico* que consideraremos tiene la siguiente premisa. Dado una ciudad representada por un conjunto V de n puntos conectados por un conjunto E de m calles unidireccionales, dos puntos críticos s y $t \in V$, y un conjunto

$$P := \{p_1 \ ... \ p_k\}$$

de k calles bidireccionales candidatas, queremos saber cuál es la mínima distancia que deberemos recorrer para llegar de s a t, dado que se construya una de estas calles.

Para ello, vamos a contar con la longitud ℓ_i^c, $1 \leq i \leq m$ de cada calle en la ciudad y la longitud ℓ_j^p, $1 \leq j \leq k$ de cada calle posible a construir.

Por ejemplo, si tuvieramos la siguiente ciudad —cuyas calles candidatas están en color gris— y los puntos críticos $s = 1$ y $t = 4$



entonces podríamos construir la calle $2 \leftrightarrow 3$ con peso 5 para lograr un camino $1 \to 2 \to 3 \to 4$ con distancia mínima 35.

1.1. Modelado como un problema de camino mínimo. A partir del ejemplo anterior, vemos que el problema del *tráfico* se puede modelar de manera intuitiva como un problema de *camino minimo* en grafos: sea D el digrafo asociado a una ciudad $(V,\ E)$ y sea $w : E \to \mathbb{R}_{\geq 0}$ una función de peso, donde $w(c_i) = \ell_i^c$, $c_i \in E$ para todo $1 \leq i \leq m$, y $w(p_i) = \ell_i^p$, $p_i \in P$ para todo $1 \leq i \leq k$. Luego, podemos resolver el problema del *tráfico* si evaluamos el camino mínimo entre s y t para cada digrafo en la sucesión

$$\{D\} \cup \{(V,\ E \cup \{e,\ \bar{e}\}) : e \in P\} \tag{1}$$

utilizando la función de peso w.

Sin embargo, esto no es eficiente. De emplear el algoritmo de *Dijkstra* con *min-heap*, la complejidad de peor caso estaría en $O(k \cdot m \log n)$. Vamos a ver cómo lo podemos mejorar.

1.2. El algoritmo. Notar primero que el camino mínimo entre dos vértices s y t satisface la propiedad de *subestructura óptima*[3]. Esto es, cada sección del camino forma, a su vez, un camino mínimo[4].

Sigue que, si $\delta_D : E \to \mathbb{R}$ es la distancia mínima entre cualquier par de vértices en un digrafo $D = (V,\ E)$ con función de peso $w : E \to \mathbb{R}$ que no tiene ciclos negativos, entonces

[3]Ver Cita 1, sección 16.2: *Elements of a greedy strategy.*

[4]Si no, podríamos reemplazar esta sección por otra de menor distancia, lo que es una contradicción.

para cualquier par de vértices s y t en V, para los cuales existe un camino $s \rightsquigarrow t$, y una arista $(u,\ v)$ perteneciente a este camino, $\delta_D(s,\ t) = \delta_D(s,\ u) + w(u,\ v) + \delta_D(v,\ t)$.

Vamos a demostrar en la siguiente sección que una consecuencia de esta observación es que, de agregar una arista $e = (u,\ v)$ a D con peso ℓ no negativo, entonces

$$\delta_{D+e}(s,\ t) = \text{mín}\{\delta_D(s,\ u) + \ell + \delta_D(v,\ t),\ \delta_D(s,\ t)\}. \tag{2}$$

En particular, dado que $\delta_D(s,\ t) = \delta_{D^t}(t,\ s)$[5] y que nuestro problema se restringe a pesos no negativos, estas observaciones nos permiten considerar el siguiente algoritmo.

```
1 proc trafico(D: (V, E), P: sec< V × V × ℝ≥0 >, w: E → ℝ≥0, s, t: V) → ℝ≥0:
2     δ^s ← camino_minimo(D, w, s)
3     δ^t ← camino_minimo(D^t, w, t)
4     μ ← δ^s(t)
5     para (u, v, ℓ) en P:
6         γ1 ← δ^s[u] + ℓ + δ^t[v]   // s ⇝ u → v ⇝ t
7         γ2 ← δ^s[v] + ℓ + δ^t[u]   // s ⇝ v → u ⇝ t
8         μ ← min{μ, γ1, γ2}
9     retornar μ
```

Algoritmo 1. Pseudocódigo para el problema del *tráfico*.

El mismo aplica un algoritmo de *camino mínimo con única fuente* sobre el grafo de entrada D, a partir de s, y sobre el grafo transpuesto D^t, a partir de t, para saber la distancia mínima —que se guarda en los diccionarios δ^s y δ^t— de ambos vértices a todo el resto de los vértices en el digrafo. Luego, aplica la ecuación 2 para determinar cuál es la distancia mínima entre s y t en cada par de digrafos $(D + e,\ D + \bar{e})$[6] para cada calle bidireccional e en P.

1.3. Demostración de correctitud. Dada la discusión anterior, basta demostrar que la ecuación 2 se satisface para demostrar que el algoritmo 1 encuentra la distancia del camino mínimo entre s y t dentro del conjunto de digrafos definido en la ecuación 1.

Demostración. Sea D un digrafo $D = (V,\ E)$ con función de peso $w : E \to \mathbb{R}$ que no tiene ciclos negativos y sea $\delta_G : E \to \mathbb{R}$ la distancia mínima entre cualquier par de vértices en un digrafo G cualquiera.

Consideremos el digrafo $D+e$, con $e = (u,\ v)$, tal que e es una arista entre dos vértices de V que no está en D y tiene peso ℓ no negativo. Si e no pertenece a ningún camino mínimo de s a t en $D + e$, sigue trivialmente que

$$\delta_{D+e}(s,\ t) = \delta_D(s,\ t)$$

ya que $D \subset D + e$. Si, en cambio, sí pertenece, entonces debe ser que

$$\delta_{D+e}(s,\ t) \leq \delta_D(s,\ t)$$

ya que, o bien e *mejora* el camino mínimo entre ambos vértices, o bien lo mantiene igual, pero no puede suceder que lo empeore. Si no, dado que cualquier camino mínimo en D está en $D + e$, el camino que contiene a e no sería mínimo.

[5]Notar que los caminos en un digrafo son *dirigidos*. Luego, las distancias son simétricas respecto al digrafo transpuesto.

[6]Esto es equivalente a considerar el digrafo $D + \{e,\ \bar{e}\}$, ya que ambas aristas no pueden pertenecer a un mismo camino. Esto se debe a que, si ambas aristas pertenecieran en simultáneo a un recorrido, entonces formarían un ciclo. Los camínos mínimos no tienen ciclos[7]

Del resultado anterior y, por la propiedad de *subestructura óptima* del camino mínimo, sigue que

$$\delta_{D+e}(s,\ t) = \begin{cases} \delta_{D+e}(s,\ u) + \ell + \delta_{D+e}(v,\ t) & \text{si } e \in\ C_{st}(D+e) \\ \delta_D(s,\ t) & \text{si no} \end{cases} \tag{3}$$

donde $C_{st} \subset D + e$ es el subgrafo de caminos mínimos entre s y t.

Dado que cualquier camino mínimo de s a u y cualquier camino mínimo de v a t en $D+e$ no puede contener a la arista e —ya que, si no, se formaría un ciclo—, podemos concluir que $\delta_{D+e}(s,\ u) = \delta_D(s,\ u)$ y $\delta_{D+e}(v,\ t) = \delta_D(v,\ t)$. En particular, sigue que la ecuación 3, es equivalente a

$$\delta_{D+e}(s,\ t) = \text{mín}\{\delta_D(s,\ u) + \ell + \delta_D(v,\ t),\ \delta_D(s,\ t)\}.$$

□

1.4. Complejidad temporal y espacial. El algoritmo `trafico` depende casi exclusivamente de la implementación de *camino mínimo* que utilicemos. Considerando que puede haber diferentes valores para los pesos de las aristas no podemos hacer BFS, y como no se sabe si en el grafo hay ciclos, tampoco se puede usar el algoritmo de orden topológico. Estos dos resultan eficientes si se garantizan sus respectivas precondiciones. Dado que el peso de las aristas es no negativo, la mejor[8] complejidad que podemos lograr corresponde a utilizar el algoritmo de *Dijkstra* sobre una estructura de *fibonacci-heap*, cuyo costo espacial es $\Theta(n)$ y cuyo costo temporal es $\Theta(m + n \log n)$[9]. Luego, nuestro problema resultaría en una complejidad temporal en $\Theta(k + m + n \log n)$, correspondiente a la construcción del digrafo de entrada y su transpuesta, dos invocaciones de *camino mínimo* y las k iteración del ciclo que comienza en la línea 5. El costo espacial estaría en $\Theta(m + n)$, correspondiente a las estructuras de los digrafos y el costo espacial de *camino mínimo*.

2. Evaluación empírica

Para revisar la diferencia en eficiencia entre distintas implementaciones de TRAFICO sobre el algoritmo de Djikstra consideramos tres algoritmos para solucionar este problema. El primero, al que llamaremos *heap*, utiliza un heap para guardar todas las aristas y luego ir sacando las candidatas. y puede extraer el mínimo en tiempo constante y cambiar una clave en un razonable $\Theta(log(n))$. La complejidad temporal de este algoritmo es $\Theta(mlogn)$[10]. Una segunda implementacion, a la que llamaremos *ingenua*, funciona de la misma manera pero implementa el heap de forma ingenua, con un arreglo de costos y otro de valores. Esto permite cambiar claves en tiempo constante, pero a cambio encontrar el mínimo se hace en $Theta(n)$. La complejidad resultante es $Theta(n^2)$[11]. Finalmente usamos *queue*, una implementacion que utiliza la cola de prioridad del *prelude* de C++ y un arreglo *marcados* que indica si un elemento ya pasó por el queue o no. En vez de agregar todos los nodos a la cola como los otros dos algoritmos, este los va colocando en a medida que aparecen en la lista de adyacencia de los nodos candidatos. El arreglo *marcados* evita que se repitan nodos. Tiene complejidad espacial de $Theta(mlogn)$[12].

[8] En base a los algoritmos conocidos.

[9] Ver cita 1, sección 24.3.

[10] Ver diapositivas de la clase 7, AED3

[11] Ver diapositivas de la clase 7, AED3

[12] Ver diapositivas de la clase práctica 12 sobre camino mínimo

Teóricamente, *ingenuo* será más eficiente para instancias *densas*, mientras que los otros dos resultan especialmente buenos para entradas *ralas*. Sin embargo los tres algoritmos tienen distintos detalles de implementación que resultan interesantes de analizar empíricamente: Si bien *ingenuo* es asintóticamente superior, realiza mucho trabajo de más al tener que recorrer la estructura entera cada vez para encontrar el mínimo. *Queue* puede recorrer el mismo nodo varias veces si se ingresó a la cola en múltiples instancias, y además no tiene el costo de cambiar claves dentro del heap, pero como queue es una estructura de *prelude* es probable que esté bien optimizada, en comparación con *heap* que fue implementada .ª mano".

Para ver empíricamente la diferencia en eficiencia entre estas tres implementaciones, realizamos una serie de evaluaciones respecto al tiempo de ejecución en función del tamaño de la entrada, primero para muestras aleatorias de tamaño $N = 15000$ $M = 15000 + 11247750 * k$ para cada k natural en el rango $1 \leq k \leq 10$, para cómo afecta la cantidad de aristas al tiempo de ejecución en cada algoritmo, luego con muestras de tamaño $N = 10000k$ para cada k natural en el rango $1 \leq k \leq 10$, para observar el desempeño en grafos ralos. Y finalmente con muestras de tamaño $N = 1000000k$ para cada k natural en el rango $1 \leq k \leq 10$, para comparar el desempeño en grafos ralos de *heap* y *queue*.

Realizamos cada evaluacion diez veces para reducir la variación de los resultados y tomamos el promedio aritmético. Para la segunda evaluación lo hicimos 100 veces, dado que los tiempos eran pequeños.

A su vez, controlamos los parámetros de la función de la siguiente forma. Elegimos $K = 0$, $S = 1$, $T = 2$, y definimos cada arista $x_i = (a_i, b_i, w_i)$, para todo $1 \leq i \leq M$, $0 \leq w_i \leq 1000$ y $1 \leq a_i b_i \leq N$ aleatorios, tal que $a_i \neq b_i$ y $(a_i, b_i) \neq (a_j, b_j)$ para todo $1 \leq i, j \leq Mei \neq j$

Notar que ninguna de estas decisiones afecta a la complejidad de las implementaciones de Djikstra. En efecto, el valor de K solo es relevante luego de haber corrido Djikstra, y S y T aleatorios no deberían tener efecto sobre la complejidad asintóticamente.

La figura 1 expone los resultados de la experimentación en grafos *densos*. Nótese que a pesar de su superioridad asintótica, *ingenuo* resulta el más lento, incluso en el último caso que ilustra un grafo completo. Quizás tiene constantes demasiado grandes. Claramente *queue* es la más rápida, quizás por la eficiencia de la implementación de la estructura.

La figura 2 expone los resultados de las experimentaciones en grafos *ralos*. Notese que el algoritmo *ingenuo* resultó extremadamente ineficiente en esta situación, por lo que se debieron utilizar instancias pequeñas para poder compararlo con los otros algoritmos. Como los tiempos de *heap* y *queue* son demasiado bajos en esta instancia, se comparan los dos solos con una entrada 100 veces más grande. De vuelta se nota una clara diferencia a favor del algoritmo *queue*.

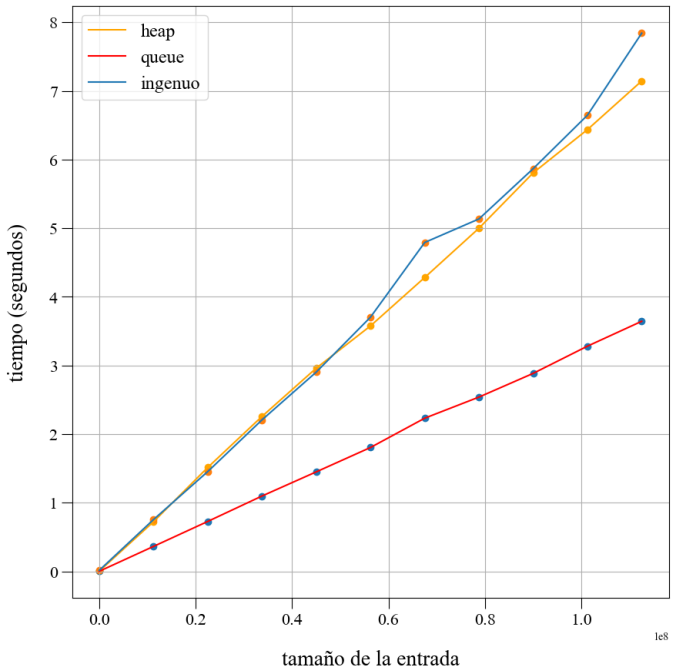


FIGURA 1. Tiempo de ejecución de TRÁFICO en función del tamaño de entrada M para el algoritmo *ingenuo*, el *heap*, y *queue*.

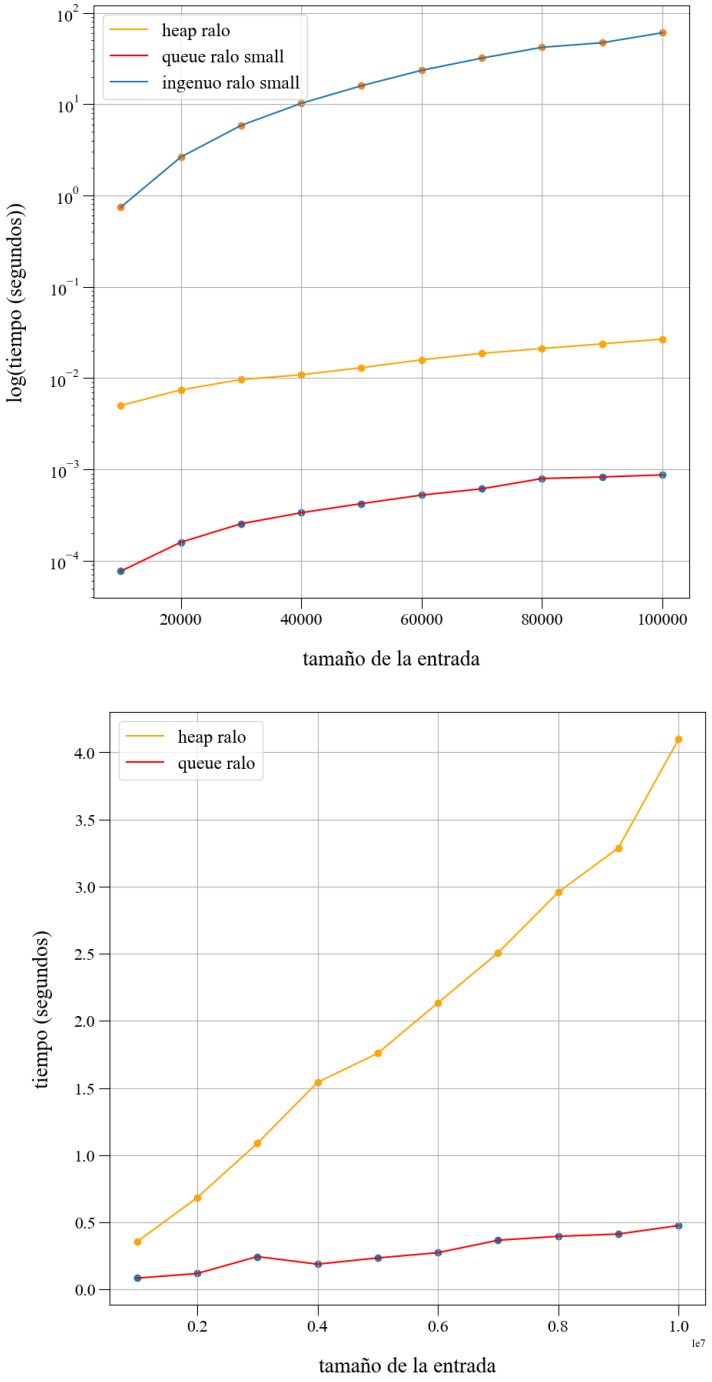


FIGURA 2. Izquierda: tiempo de ejecución de TRAFICO en función del tamaño de entrada N para el algoritmo *ingenuo*, el *heap*, y *queue*. Derecha: ejecución de TRÁFICO en función del tamaño de entrada N con valores grandes, para *heap* y *queue*.