Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Informe del problema "Mejorando el tráfico"

Integrante	LU	Correo electrónico
Fabian Gómez Laguna	522/20	fabiangl2011@gmail.com
Nicolás Sergio Aquino	290/20	aquino_nicolas@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Presentación y descripción del problema.

El problema sobre el que se basará este informe trata de reducir la longitud del camino más corto entre dos puntos s y t de una ciudad eligiendo una calle apropiada bidireccional para ser construida de un conjunto de candidatos. Más específicamente, dado el mapa de una ciudad con n puntos numerados del 1 al n, y con m calles *unidireccionales* que conectan pares de puntos, se proponen k calles *bidireccionales* como candidatas a ser construidas y se quiere escribir un programa para elegir una calle de la lista propuesta, con el fin de reducir la longitud del camino más corto entre dos puntos críticos diferentes s y t.

Para simplificar el problema, no se pide saber cuál calle se debe construir, sino que solo se pide la longitud más pequeña posible del camino más corto después de construir la calle bidireccional elegida de la lista propuesta. Además, en caso de que no exista un camino desde s hasta t, se pide indicarlo devolviendo -1.

2. Explicación del algoritmo

Llamemos $G = (V, E)$ al grafo pesado que representa el mapa de la ciudad, en donde los n puntos los representamos con n vértices y cada una de las m calles *unidireccionales* $c_i = (u_i, v_i)$ con longitud $l((u_i, v_i)) = l_i$, $1 \leq i \leq m$, los representamos con una arista que conecta el vértice u_i con el vértice v_i con peso l_i, donde $l : E \to \mathbb{Z}_{\geq 0}$ es la función peso de las aristas. Para que tenga más sentido, en el informe diremos *longitud* cuando nos referimos al *peso* de una arista.

En principio, nos gustaría saber el camino mínimo desde s hasta t en G para tener un punto de referencia con el cual comparar las distancias al probar las calles candidatas. Pero no solo nos interesa eso, ya que para saber si construir la calle *bidireccional* $b_j = (u_j, v_j)$ con longitud $l((u_j, v_j)) = q_j$, $1 \leq j \leq k$, mejora la distancia desde s hasta t, debemos poder saber la longitud del camino más corto desde s hasta u_j (o v_j), así como también desde v_j (o u_j) hasta t, para así concluir si pasar por esa nueva calle (en alguno de los dos sentidos) mejora la longitud más pequeña posible obtenida hasta el momento.

Teniendo en cuenta eso, nos interesa entonces obtener las longitudes de los caminos mínimos desde s hacia todos los vértices, y de las longitudes de los caminos mínimos desde todos los vértices hacia t. Observemos que para lo segundo, también obtendríamos las longitudes de los caminos mínimos desde t hacia todos los vértices, si se fuese en sentido inverso. Podemos saber esas longitudes utilizando alguno de los algoritmos vistos en la materia. En particular, como sabemos que las longitudes son todas no negativas, podemos utilizar el algoritmo de Dijkstra, que calcula justamente estas longitudes que buscamos.

Para calcular las longitudes de los caminos mínimos de t hacia todos los vértices, armamos un grafo $G_T = (V, E_T)$ que es igual a G pero con las direcciones de las aristas invertidas, es decir que para toda arista dirigida $(u \to v) \in E$ tenemos que $(v \to u) \in E_T$.

Una vez obtenidas las distancias utilizando Dijkstra, lo que queda es ir probando una por una las calles candidatas b_j y obtener la longitud mínima que se obtendría para el nuevo camino mínimo desde s hasta t si se pasa por la nueva calle, probando en ambos sentidos, y nos quedamos con la mínima longitud obtenida luego de probar todas las calles. Finalmente nos quedamos con el mínimo de esta longitud obtenida y la que ya habíamos calculado en G originalmente. Esta será la longitud que buscamos. Si después de hacer todo esto tenemos que $res = \infty$, entonces no hay camino de s a t, con lo cual podemos devolver res y luego devolver -1 en otra función.

El pseudocódigo del algoritmo es el siguiente:

entrada: G y G_T, dos vértices s y t, $l : E \to \mathbb{Z}_{\geq 0}$ y un conjunto de calles candidatas B
salida: res la longitud más pequeña posible del camino más corto después de construir la calle bidireccional elegida de B

$\pi_1 \leftarrow \texttt{Dijkstra}(G, s)$
$\pi_2 \leftarrow \texttt{Dijkstra}(G_T, t)$
$res \leftarrow \pi_1[t]$
for all $b = (u, v) \in B$ **do**
  $d_1 \leftarrow \pi_1[u] + l((u, v)) + \pi_2[v]$
  $d_2 \leftarrow \pi_1[v] + l((u, v)) + \pi_2[u]$
  if $res >$ mín(d_1, d_2) **then**
    $res \leftarrow$ mín(d_1, d_2)
  end if
end for
return res

3. Justificación de correctitud del algoritmo

Por Dijkstra, sabemos que π_1 contiene las longitudes de los caminos mínimos desde s hacia todo vértice $u \in G$, con longitud $\pi_1[u]$, y que π_2 contiene las longitudes de los caminos mínimos desde t hacia todo vértice $v \in G_T$, con longitud $\pi_2[v]$, o lo que es lo mismo, π_2 contiene las longitudes de los caminos mínimos desde todo vértice $v \in G$ hacia t, con longitud $\pi_2[v]$. Si definimos la función $d(\cdot, \cdot)$ como la función que indica la longitud de un camino mínimo entre dos vértices, tenemos en particular que $d(s, u) = \pi_1[u]$ y que $d(v, t) = \pi_2[v]$ $\forall u, v \in V(G)$.

Antes de entrar al ciclo, vale que $res = d(s, t)$, que es la longitud del camino mas pequeño posible si no se elige ninguna calle candidata b. Veamos que al finalizar cada iteración i, el valor de res se actualiza si la longitud del camino mínimo de s a t mejora al elegir construir la calle bidireccional b_i.

Al iniciar la iteración i se elige a la calle candidata $b_i = (u_i, v_i)$ y se calculan d_1 y d_2 los cuales representan la nueva longitud mínima de s a t si forzamos a pasar por b_i por una dirección o por la otra respectivamente. Para ello, tomamos $d(s, u)$ y $d(v, t)$ para d_1 para asegurarnos de tomar la mejor ruta a las entradas de la calle y sumamos $l((u, v))$. Para d_2 es análogo. De estas dos distancias, nos quedamos con el mínimo.

Si ese mínimo es mejor que lo que habíamos calculado en res, actualizamos res con ese mínimo. Caso contrario lo ignoramos.

Al finalizar el ciclo tendremos en res la mínima longitud que se puede obtener de un camino mínimo de s hacia t considerando las calles candidatas B.

4. Complejidad

La complejidad del cuerpo del ciclo es $\mathcal{O}(1)$ ya que son solo sumas, asignaciones y comparaciones con números enteros. En total se hacen $|B|$ iteraciones en el ciclo con lo cual el ciclo es

$\mathcal{O}(|B|) = \mathcal{O}(k)$. Asignar $res \leftarrow \pi_1[t]$ es también $\mathcal{O}(1)$ dejando la complejidad del algoritmo como $\mathcal{O}(k) + \mathcal{O}(\texttt{Dijkstra})$, con lo cual queda analizar la complejidad de Dijkstra.

Dependiendo de la implementación para buscar el siguiente vértice u sin procesar con el mínimo $\pi[u]$ dentro del algoritmo de Dijkstra, podemos obtener distintas complejidades. Por ejemplo, podemos obtener un algoritmo $\mathcal{O}(n^2)$ si se busca linealmente el vértice dentro de π, $\mathcal{O}(m \log(n))$ si se utiliza un min-heap como estructura de datos para hacer lo mismo, o también $\mathcal{O}(n \log(n) + m)$ si se utiliza un Fibonacci Heap.

El enunciado del problema nos indica las cotas de las entradas, siendo $n \leq 10^4$ y $m \leq 10^5$. Como $n^2 \leq 10^8$, en peor caso sabemos que $m \ll n^2$, con lo cual podemos pensar G como un grafo ralo y entonces la mejor implementación en teoría para Dijkstra es una implementación para ralos, siendo lo ideal usar un Fibonacci Heap como su estructura de datos. Por lo tanto, la mejor complejidad teórica del algoritmo queda en $\mathcal{O}(n \log(n) + m + k)$.

5. Experimentación

Si bien en teoría la mejor implementación para Dijkstra es usando un Fibonacci Heap, este es difícil de implementar, con lo cual vamos a utilizar como nuestra implementación para ralos un min-heap. Este se puede implementar en C++ utilizando estructuras de datos de la stdlib: `set` y `priority_queue`. El primero está implementado sobre *red-black trees* y el segundo sobre *heaps*.

Lo malo del segundo es que no permite modificar los valores que ya están dentro de su heap, con lo cual una forma de poder relajar las aristas es agregando nuevamente las longitudes *candidatas* $\pi'[u]$ al heap y al momento de sacarlo de la `priority_queue` chequear si $\pi'[u] > \pi[u]$, ya que si eso pasa, entonces el valor más grande es el valor viejo, con lo cual descartamos ese. Esto causa que la complejidad con un `priority_queue` sea $\mathcal{O}(m \log m)$, ya que en peor caso tenemos todas las aristas en el min-heap al realizar la relajación.

También vamos a ver cómo se comporta la implementación de Dijkstra que funciona mejor para grafos densos, el cual es simplemente la que busca linealmente el mínimo en π en cada iteración, haciendo que la complejidad sea $\mathcal{O}(n^2)$, y lo compararemos con los resultados de la implementación optimizada para ralos.

5.1. Resultados

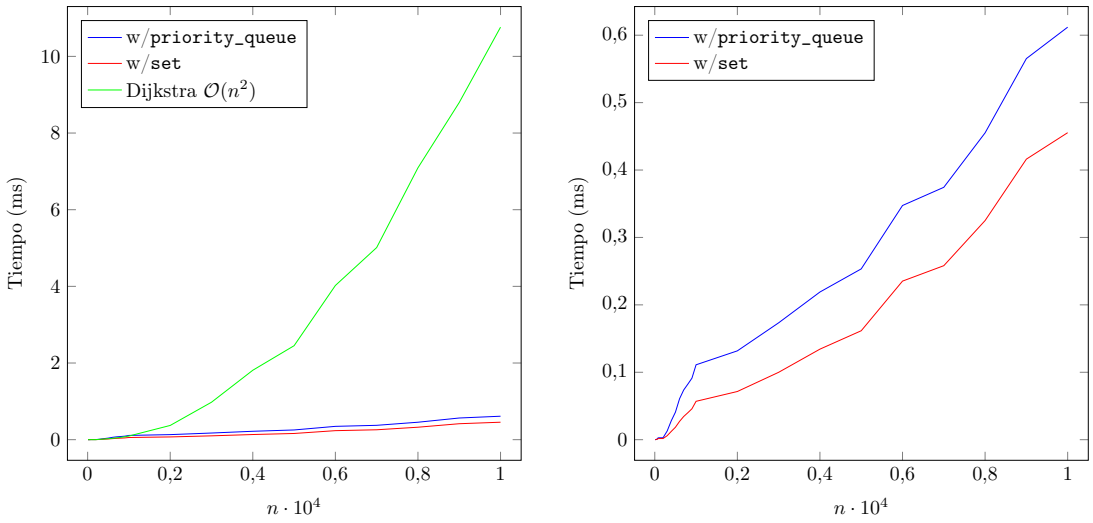

Figura 1: Comparación de performance con entradas generadas aleatoriamente

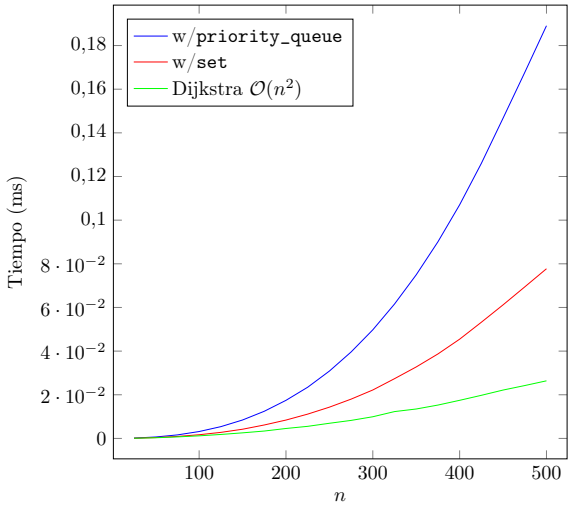

Figura 2: Comparación de performance con grafos densos

5.2. Conclusiones

Podemos observar en la figura 1 que efectivamente, la mejor implementación para este problema es utilizando una optimización para grafos ralos, ya que al utilizar la versión de Dijkstra para densos empeora mucho la performance para n grandes. Además, observando la comparación entre `priority_queue` y `set`, se puede ver que utilizar un `set` es ligeramente mejor que utilizar la `priority_queue`, dando a entender que es mejor utilizar un `set` para nuestra implementación de Dijkstra.

También podemos ver en la figura 2 que para instancias en donde $m \approx n^2$, la implementación de Dijkstra para densos resultó ser mejor en términos de performance que las otras implementaciones, lo cual concuerda con nuestras suposiciones anteriores.

Concluimos entonces que para este problema es preferible usar una implementación de Dijkstra optimizada para ralos, a pesar de que para algunas instancias específicas la versión para densos pueda dar una ligera mejora.

Referencias

Dijkstra - finding shortest paths from given vertex. URL: https://cp-algorithms.com/graph/dijkstra.html#implementation.

Dijkstra on sparse graphs. URL: https://cp-algorithms.com/graph/dijkstra_sparse.html.