Victoria Klimkowski y Lorenzo Gandolfo

Introducción

Este informe describe la resolución de un problema de construcción de calles bidireccionales en una ciudad que sólo cuenta con calles unidireccionales para minimizar la distancia entre un punto inicial y otro final dados. Este problema se modela utilizando un grafo dirigido y 2 implementaciones distintas del algoritmo de Dijkstra (algoritmo para encontrar el camino mínimo) para comparar los tiempos de ejecución.

Resolución

En una ciudad de N puntos y M calles unidireccionales se quiere construir una calle bidireccional entre la K posibles para minimizar la distancia entre 2 puntos dados S (inicial) y T (final). Cada calle tiene una distancia L.

La estrategia consiste en generar 2 grafos (G1 y G2) exactamente iguales con todas las calles unidireccionales de la ciudad. Ambos grafos están conectados mediante las K calles bidireccionales propuestas. Esto es, si hay alguna calle bidireccional que conecta el nodo p con el nodo q, en realidad se conectará[1] al nodo p∈G1 con el nodo q∈G2 y al nodo q∈G1 con el nodo p∈G2). Es importante resaltar que esta conexión no es bilateral. Es decir, las calles bidireccionales son las que nos permiten ir de G1 a G2 pero una vez en G2 no hay manera de retornar a G1, por lo tanto no se conectará al nodo p∈G2 con el nodo q∈G1 ni tampoco el nodo q∈G2 con el nodo p∈G1. Esto es así porque se puede construir como mucho una calle bidireccional. Entonces, si nos encontramos en G2 significa que ya tomamos una calle bidireccional y no podemos tomar otra.

Se puede decir que entonces tenemos dos opciones: o bien tomamos una calle bidireccional o bien no tomamos ninguna.

Recordemos que queremos encontrar la longitud del camino de S a T. Luego, si queremos encontrar la distancia sin tomar calles bidireccionales debemos correr Dijkstra desde S∈G2 al T∈G2. En cambio, si queremos encontrar el camino de S a T pasando por una calle bidireccional debemos correr Dijkstra desde S∈G1 al T∈G2 y nos quedamos con la más chica.

Puede pasar que ambas sean infinito, lo cual significa que no hay camino posible de S a T.

Complejidad

Para cada caso C creamos el grafo G, el cual tiene V vértices (con V=2*N+2) y E aristas (con E=2*M+2*K). La creación del grafo es O(V+E)

Como ya dijimos antes, la duplicación de los nodos y aristas no cambia la complejidad ya que 2 es una constante. Luego se corre Dijkstra 2 veces (de S∈G1 al T∈G2 y de S∈G2 al T∈G2) y se devuelve el mínimo entre ambos resultados.

Para realizar Dijkstra, primero se inicializan dos vectores de N elementos que representarán los nodos visitados y las distancias a los demás nodos. Esto es O(V).

También tendremos una cola de prioridad para almacenar las distancias desde nuestro nodo S a todos los demás. La inserción y la eliminación en un heap es O(log V). El primero en agregarse es el mismo nodo S.

---

[1] En este contexto, decimos que “conectar p con q” es que haya una arista unidireccional de p a q.

El bucle principal se realiza hasta que la cola esté vacía, es decir que se ejecutará O(V) veces. En cada iteración se extrae el nodo con la menor distancia de la cola de prioridad lo cual será O(log V).
Se verifica en cada paso que el nodo extraído no haya sido visitado, por lo cual cada nodo se visitará una vez.
Para cada nodo extraído U se relajan las aristas que lo conectan con los nodos vecinos (nodos V). Para esto hay que chequear que dist(S,U)+dist(U,V) < dist(S,V) y que el nodo vecino V no haya sido visitado para agregarlo a la cola de prioridad. Es decir, todas las aristas se van a visitar O(E) veces y la inserción a la cola de prioridad es O(log V).
Entonces, la complejidad de Dijkstra es O((V+E)* log V) y todo el algoritmo que modela el problema será O(C* (V+E+ (V + E)* log V))). En términos del tamaño de entrada quedaría O(C* (N+M+K+ (N+M+K)* log N)))

Otra implementación de Dijkstra

La complejidad de este algoritmo es de O(V^2 + E).
Primero se inicializan los vectores que representan las distancias a cada nodo y los nodos visitados O(V).
Luego, hay 2 bucles for anidados para encontrar el nodo no visitado con la distancia mínima O(V^2).
Una vez encontrado, se itera sobre los vecinos para realizar la relajación de las aristas y actualizar la distancia mínima en caso de ser necesario. La relajación ocurre O(E) veces.
Entonces, la complejidad del problema pasa a ser de O(C* (V+E+ V^2)). En términos del tamaño de entrada quedaría O(C* (N+M+K+ N^2))

Prueba empírica

Realizamos un gráfico comparativo con 99 casos de test. Tomamos distintos valores de N equiespaciados entre 100 y 10000. La cantidad de aristas unidireccionales para cada N la tomamos como el min(8*N , N*N / 2) y la cantidad de aristas bidireccionales la tomamos como el min(N , 300). Los pesos de las aristas son números random entre 1 y 1000.
Luego, para cada caso de test corrimos lo corrimos para ambas implementaciones de Dijkstra para hacer un gráfico comparativo de las complejidades.

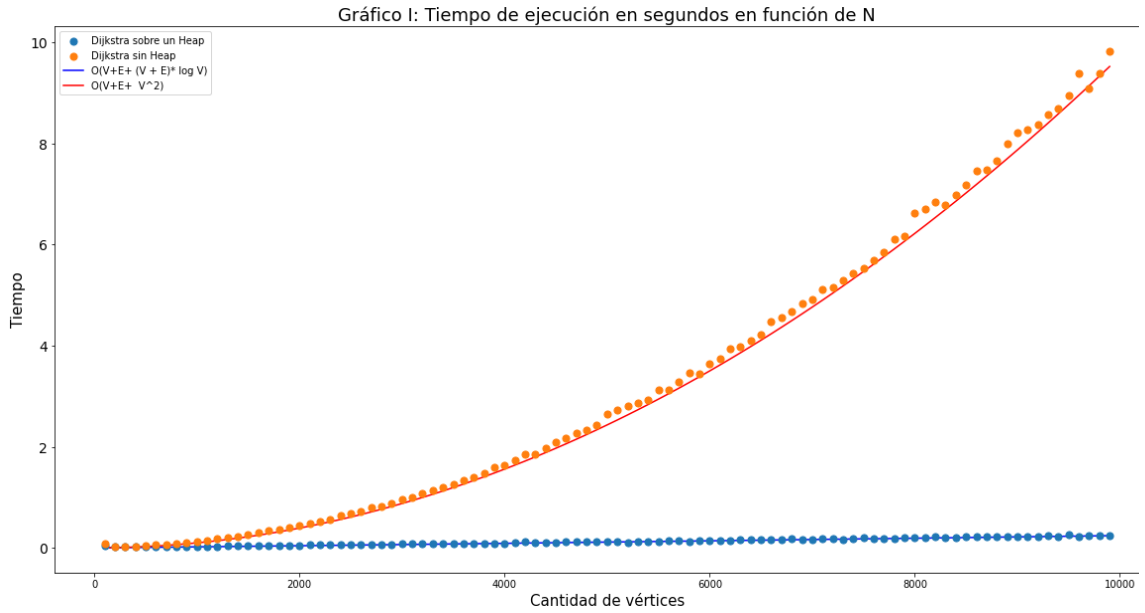

*Grafico I: plot comparativo de tiempo de Dijkstra y ajustes de complejidad respectivamente para M = min(8*N , N*N / 2) y K = min(N , 300).*

Se puede apreciar la tendencia temporal creciente sobre todo en la implementación que no utiliza la cola de prioridad. También puede verse claramente el buen ajuste a la complejidad obtenida teóricamente. Para la línea roja literalmente tomamos la función (N+M+K+ (N+M+K)*log(N)) es decir la complejidad teórica obtenida para Dijkstra sin heap y a ojo fuimos viendo si existía alguna constante (1/3800000) por la que multiplicar a esa función que ajustara bien los valores de la implementación de Dijkstra sin heap. Análogamente para Dijkstra con heap (cuya constante fue 1/10600000).
Luego lo corrimos para M un número random entre 1 y min(8*N , N*N / 2) en lugar de ser exactamente la cota superior. Análogamente para K siento este un número random entre 1 y min(N,300).

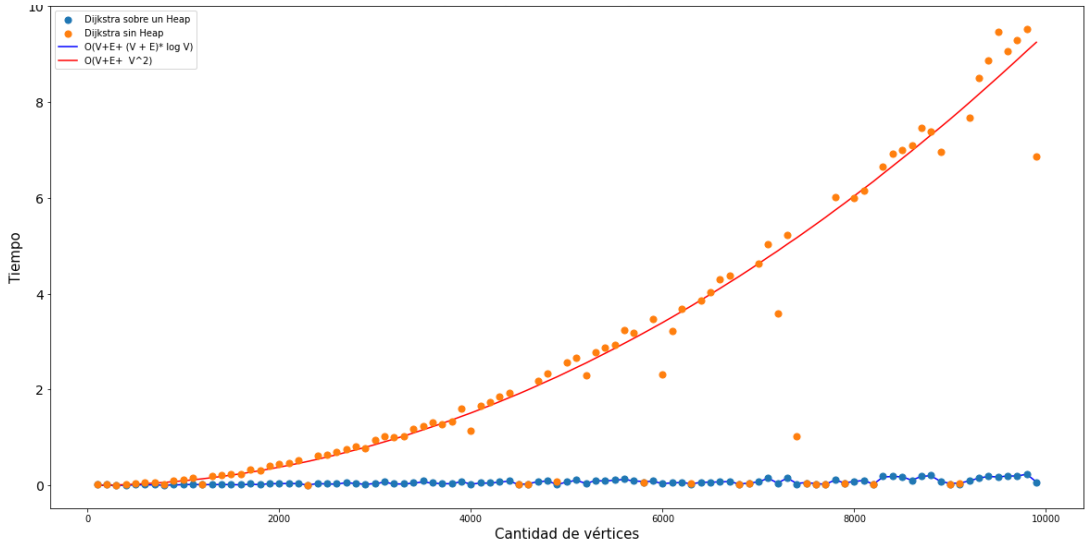

*Grafico II: plot comparativo de tiempo de Dijkstra y ajustes de complejidad respectivamente para algún 1 < M < min(8*N , N*N / 2) y 1 < K < min(N , 300).*

En este gráfico se pueden ver más cantidad de “outliers” debido a que justamente la complejidad depende de la cantidad de vértices y la de aristas. Sólo la mostramos en función de los vértices y en el gráfico anterior las aristas dependían más fuertemente de ellos.
En ambos grafos mostramos la función de ajuste de complejidad obtenida teóricamente multiplicada por una alguna. Probamos con varias constantes para ver si había alguna que ajustara mejor y obtuvimos 1/3800000 para Dijkstra sobre un heap y 1/10600000 para la otra.

```
dijkstraCONHeap = Ns + Ms + (Ns + Ms)*(np.log(Ns))
dijkstraSINHeap = Ns + Ms + (Ns*Ns)

plt.scatter(Ns, tiempos_ejec_dsu_opt, label='Dijkstra sobre un Heap', s=50, zorder=2)
plt.scatter(Ns, tiempos_ejec_dsu_sin_opt, label='Dijkstra sin Heap', s=50, zorder=2)

plt.plot(Ns, dijkstraCONHeap / 3800000, label='O(V+E+ (V + E)* log V)', linestyle='-',
color='b', zorder=1)
plt.plot(tamaños, dijkstraSINHeap / 10600000, label='O(V+E+ V^2)', linestyle='-', color='r',
zorder=1)
```

Ambos gráficos anteriores son útiles para notar que es una complejidad que crece mucho más rápidamente que al implementar Dijkstra sobre una cola de prioridad en estas instancias. Sin embargo, no se logra apreciar el detalle del ajuste de la complejidad utilizando un heap por la escala.

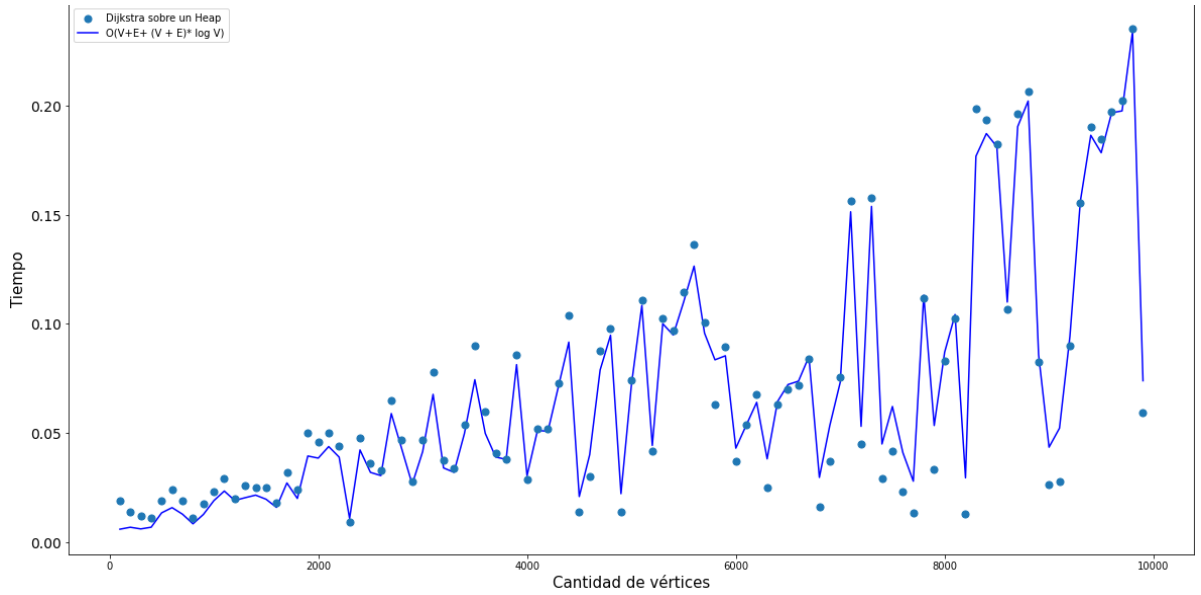

Grafico III: plot de tiempo de Dijkstra con heap y ajuste de complejidad.

En este podemos ver que la tendencia temporal creciente sigue estando presente y que la función de la complejidad O(V+E+ (V + E)* log V)) sigue siendo un buen ajuste. La manera de plotear fue la misma que antes donde definimos la función de complejidad y elegimos alguna constante (la misma) que ajuste. Lo que nos permitió ver este gráfico es que aunque la función no parezca logarítmica en su dibujo sí lo es ya que la función de complejidad (línea contigua azul) toma en cuenta el valor pertinente de los vértices y las aristas y acompaña bastante a los puntos obtenidos empíricamente.
También hicimos una experimentación para grafos densos donde pudimos ver que a medida que N tomaba valores más grandes tomaba menos tiempo usando la versión sin la cola de prioridad..

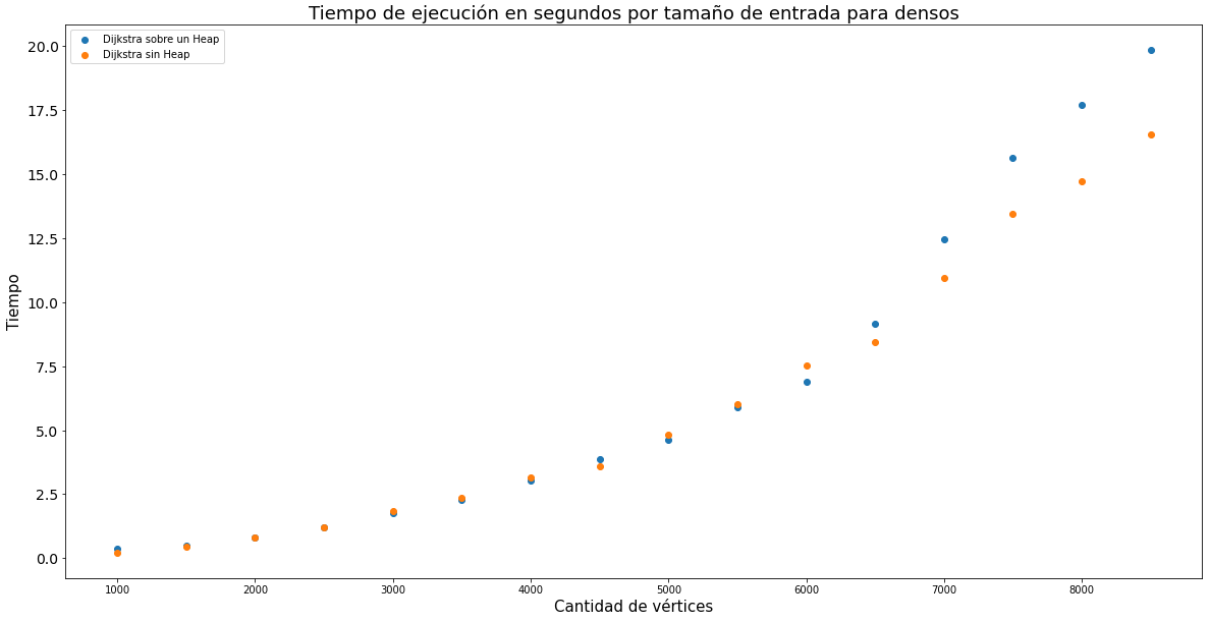

Gráfico IV: plot de 2 implementacions de Dijkstra para grafos densos.

Justificación del algoritmo

Para verificar que el algoritmo es correcto podemos ver que todo camino válido en la ciudad se corresponde con un camino en el grafo G.

Sea C = $s,v_1,v_2,...,v_n,t$ un camino válido de esquinas, existen dos posibles casos: que no se utilice ninguna arista bidireccional o que se utilice una sola.

Si el camino no contiene ninguna de las aristas bidireccionales, entonces se corresponde directamente con un camino en $G1 \subset G$ ya que por cada esquina G1 tiene un vértice y están todas las aristas unidireccionales y ninguna bidireccional. Entonces C se puede construir con las aristas en G1 y existe una correspondencia uno a uno con vértices del grafo G1 y los puntos de la ciudad. Por eso podemos correr Dijkstra entre $s \in G1$ y $t \in G1$.

En cambio, si el camino contiene una arista bidireccional entonces $C = s,v_1,v_2,p,q,...,v_n,t$. donde *pq* es la arista bidireccional utilizada. Luego, como en G tenemos cada esquina duplicada, existe un camino C' que se corresponde con C: $C' = s,v_1,v_2,p,q',...,v'_{n-1},v'_n,t$. donde s,v_1,v_2,p pertenecen a G1 y $,v'_{n-1},v'_n,t$ pertenecen a G2.

Ahora, sea un camino P en G, demostrar que existe un camino de esquinas válido en la ciudad que se corresponde con P es análogo ya que en G se tienen modeladas a todas las esquinas y a todas las aristas bidireccionales.

Luego, como todo camino original de esquinas se corresponde con un camino en G, en particular todo camino mínimo tiene su equivalente en G y correr Dijkstra en G devuelve el camino mínimo de nuestro problema.

Conclusión

Se pudo ver teóricamente que implementar Dijkstra utilizando una cola de prioridad sobre un heap resulta en una complejidad de $O((V + E)* \log V))$ mientras que su versión "naive" sin optimizaciones resulta en una complejidad de $O(V^2 + E)$.

Además, a través de la prueba empírica, se logró visualizar el impacto que tiene en el tiempo de ejecución la elección de una implementación por sobre la otra.