DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3:
Camino mínimo y Flujo máximo

Algoritmos y Estructuras de Datos III

VLakTracking

Integrante	LU	Correo electrónico
Lakowsky, Manuel	511/21	mlakowsky@gmail.com
Vekselman, Natán	338/21	natanvek11@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

1. Introducción

Mejorando el tráfico es un problema en el cual buscamos acortar la distancia entre dos puntos de una ciudad. El mapa de ésta consta de n puntos diferentes unidos de a pares por m calles unidireccionales. Luego, dados dos puntos críticos s y t, se propone una lista de k nuevas calles bidireccionales a construir con el propósito de reducir la longitud del camino más corto entre ambos. El objetivo de este problema es encontrar de entre ellas la que minimice la distancia resultante del camino mínimo entre s y t.

Entonces, dadas las m calles unidireccionales que unen a los diferentes n puntos de la ciudad, los puntos críticos s y t, y las k calles bidireccionales propuestas, buscamos encontrar la mejor de ellas y devolver la longitud del camino mínimo resultante entre s y t luego de construirla. En caso de no existir un camino entre ambos puntos críticos, se debe de devolver el valor -1 que lo indica[1].

2. Resolución

Tenemos k calles bidireccionales a comparar y buscamos de entre ellas la que minimice el valor del camino más corto entre s y t luego de construirla. Este problema puede ser modelado con un digrafo, donde todo par de puntos está conectado por a lo sumo una arista dirigida. Luego, s y t contarán, o no, con un camino mínimo original el cual buscamos mejorar, o crear, agregando las aristas correspondientes a las calles bidireccionales propuestas. Proponemos entonces el siguiente algoritmo:

1. Armamos un digrafo que modele a la ciudad, donde cada punto es un nodo y cada calle unidireccional es una arista.
2. Utilizamos el algoritmo de Dijkstra para encontrar los caminos minimos desde s hacia todos los nodos.
3. Dando vuelta las aristas, utilizamos el algoritmo de Dijkstra para encontrar el camino mínimo de todos los nodos hacia t en el digrafo original.
4. Por cada arista bidireccional k_i de largo $l(k_i)$ que une a dos puntos v y w, y sea $d(u, p)$ la longitud del camino más corto entre los vértices u y p, comparamos el mínimo de entre:
   - La longitud del mejor camino hasta ahora entre s y t.
   - Ir desde s hasta v, viajar a w y desde w hasta t con costo $d(s, v) + l(k_i) + d(w, t)$.
   - Ir desde s hasta w, viajar a v y desde v hasta t con costo $d(s, w) + l(k_i) + d(v, t)$.
5. Devolver la longitud mínima de entre todas las halladas.

La idea detrás del algoritmo es considerar todas las calles posibles a construir sin tener que recalcular el nuevo camino mínimo entre s y t con el algoritmo de Dijkstra por cada una. Entonces, luego de obtener los mejores caminos desde s hacia todos los nodos, y desde todos los nodos hacia t, podemos rápidamente obtener la longitud del nuevo camino más corto luego de construir la calle k_i con las fórmulas mencionadas en el paso 4.

[1] Restricciones a los parámetros:
$n \leq 10000, m \leq 100000, k \leq 300, 1 \leq s, t \leq n, 0 < l_i, q_i \leq 1000$, con l_i y q_i la longitud de las calles de la ciudad y de las calles propuestas respectivamente.

2.1. Implementación

Presentamos a continuación una posible implmentación de la solución explicada:

Algoritmo 1: Pseudocódigo

```
struct arista {int u, v, d}
proc trafico(in s, t: int, in vector<vector<arista>> D,
             in vector<aristas> posibles) {

    // dijkstraT interpreta aristas al reves
    distS := dijkstra(D,s)
    distT := dijkstraT(D,t)

    cm := distS[t]
    for (arista k: posibles) :
        cm := min(cm, distS[k.u] + k.d + distT[k.v])
        cm := min(cm, distS[k.v] + k.d + distT[k.u])

    res := cm
    if (res = INF) res := -1
}
```

2.2. Demostración de Correctitud

Queremos ver que el algoritmo devuelve una solución válida y óptima. Por un lado, buscamos probar que la solución que obtiene es uno de los caminos mínimos entre s y t luego de agregar alguna de las k calles bidireccionales posibles a la ciudad. Por otro lado, queremos ver que no existe un resultado mejor. Es decir, para cualquier otro camino mínimo entre s y t con el agregado de una calle a la ciudad, su longitud sera igual o mayor a la devuelta.

Demostremos todo esto haciendo inducción sobre el invariante del algoritmo. Observando el mismo, se puede apreciar que luego de i pasos, se habrán considerado las primeras i calles posibles a construir, y se tiene almacenada la longitud del camino más corto entre s y t luego de haber construido una de ellas. Llamemos la variable que guarda este valor cm.

Caso base: Cuando $i = 0$, no se consideró todavía ninguna calle bidireccional y cm almacena el mejor camino entre s y t en el digrafo original. Entonces, se cumple triviálmente la hipótesis inductiva.

Paso Inductivo: Supongamos que luego de i pasos del algoritmo se consideraron las primeras i calles bidireccionales, y se tiene almacenada la longitud del mejor camino entre s y t habiendo construido una de ellas. Buscamos probar que, luego del paso $i + 1$, se consideró la siguiente calle y se actualizó correctamente el valor almacenado.

En este paso, se considera la calle bidireccional k_{i+1} que tiene longitud $l(k_{i+1})$ y conecta a los puntos v y w. Luego, el algoritmo toma la menor de entre las siguientes tres longitudes posibles:

- La ya almacenada en cm.
- Ir desde s hasta v con longitud mínima, viajar a w, y desde w hasta t con longitud mínima, con costo $d(s, v) + l(k_{i+1}) + d(w, t)$.
- Ir desde s hasta w con longitud mínima, viajar a v, y desde v hasta t con longitud mínima, con costo $d(s, w) + l(k_{i+1}) + d(v, t)$.

De ser cm el menor de los tres, por hipótesis inductiva esta es la longitud del mejor camino mínimo entre s y t hasta el momento, con lo cual se cumple que cm sigue siendo el mejor luego de $i+1$ pasos.

Ahora, pensemos que pasa cuando el mínimo es uno de los dos nuevos caminos que pasan por la calle bidireccional k_{i+1}. Supongamos que la longitud del camino que pasa por k_{i+1} desde v hacia w es la mínima de las tres opciones (la demostración es idéntica en el caso en que pase por k_{i+1} desde w hacia v).

Por hipótesis inductiva, como cm es la longitud del mejor camino mínimo hasta ahora, este nuevo camino es mejor que todos los anteriores. Luego, solo queda ver que en el digrafo donde se agregaron las aristas $v \to w$ y $w \to v$, este nuevo camino es efectivamente uno mínimo entre s y t. Para ello, debemos demostrar que no existe otro camino entre ambos puntos críticos que tenga menor longitud que $d(s,v) + l(k_{i+1}) + d(w,t)$.

Como este camino es mejor que todos los anteriores hasta este momento, en particular es mejor que el camino mínimo entre s y t del digrafo original. Entonces, como se agregaron solo las dos aristas nuevas $v \to w$ y $w \to v$, todo nuevo camino mínimo entre s y t mejor que el original deberá necesariamente pasar por alguna de ellas. Pero de existir uno mejor que los mencionados, significa que iría desde s o t hasta v o w con menor longitud que la mínima posible, lo cual es absurdo porque estas distancias fueron calculadas con el algoritmo de Dijkstra, y por ende no existen caminos más cortos. Luego, el camino $s \to v \to w \to t$ es mejor que todos los anteriores y es el mínimo en el digrafo del paso actual, con lo cual se cumple que es el mejor luego de $i+1$ pasos.

Queda entonces demostrado que en el paso k_{i+1} se consideran las primeras $i+1$ calles bidireccionales y se almancena en cm la longitud del mejor camino mínimo entre s y t habiendo construido una de ellas. Al finalizar el algoritmo, se habrán considerado todas las calles propuestas y se devolverá efectivamente la longitud del camino mínimo mas corto posible. También observar que, de no existir camino entre s y t incluso agregando algunas de las k aristas, el mínimo calculado en cada paso será siempre infinito, y por ende el algoritmo reconocerá correctamente que debe de devolver -1.

2.3. Análisis de la Complejidad

La complejidad del algoritmo presentado es la siguiente:

- Obtener input $\to O(n)$
- Generar grafo completo $\to O(n^2)$
- Ordenar aristas con QuickSort $\to O(n^2 log(n^2))$
- Algoritmo de Kruskal $\to O(n^2 \alpha(n))$

Sabemos que $\alpha(n)$ crece más lentamente que $log(n)$ por lo que la complejidad total del algoritmo es $O(n^2 log(n^2))$.

Una posible optimización al algoritmo, observando que la parte computacionalmente más costosa de este es aplicar QuickSort, es aplicar BucketSort agrupando las aristas por peso de la siguiente manera.

Aprovechando las restricciones de los parámetros del enunciado, notamos que la distancia máxima entre dos vértices es ~ 30000. Esta se da cuando uno de los vértices se ubica en una esquina del rango posible y el otro en la esquina opuesta. Por ejemplo, $x_1 = (-10000,\ -10000) \ \wedge \ x_2 = (10000,\ 10000)$.

Agrupamos las aristas de la siguiente forma. Inicialmente, elegimos un número arbitrario llamado k, y generamos un vector de $30000/k$ posiciones de tal forma que en la posición i-ésima del vector ubicamos las aristas que tienen una distancia que pertenecen al rango $[k \cdot i,\ k \cdot (i+1))$. Como la distancia esta acotada por 30000 y cada rango es disjunto, sabemos que cada arista estará ubicada en exactamente uno de los rangos posibles. Nosotros elegimos $k = 100$ arbitrariamente.

Agrupando las aristas de esta manera, sabemos que para toda arista que se halla en el i-ésimo bucket, su peso será menor al de cualquiera de los buckets mayores a i.

Luego ordenamos las aristas de la primer posición del vector y empezamos a aplicar Kruskal con las primeras aristas ordenadas. Cuando ya analizamos todas las de la primer posición, ordenamos las de la segunda posición y seguimos aplicando kruskal, así sucesivamente hasta acabar con el algoritmo. De esta manera, nos evitamos ordenar la mayoría de las aristas del grafo.

Cabe aclarar que esta optimización no mejora la complejidad del algoritmo pero, en la práctica, si mejora considerablemente la constante.

Para medir los tiempos de cómputo, por cada $N \in [100, 200, ..., 2000]$ generamos 100 tests aleatorios y calculamos el promedio del tiempo de ejecución de estos. Como se puede apreciar en el gráfico, a medida que se aumenta el tamaño del input, el algoritmo crece de la forma mencionada, aunque BucketSort tiene una curva de crecimiento más baja.[2]

[2]Todos los tests fueron ejecutados desde el mismo ordenador para todos los algoritmos, con un CPU Intel Core i5 6400 (4-core) y 16gb de RAM DDR4 (1066MHz).

2.4. Implementaciones de Kruskal

Observando que el verdadero costo computacional de nuestro algoritmo proviene de realizar Kruskal (y el correspondiente ordenamiento de las aristas), podemos experimentar con diferentes implementaciones y optimizaciones del mismo para buscar una mayor eficiencia. En nuestra versión original del algoritmo realizamos un Kruskal con un DSU semi-optimizado con Path Compression, pero existen más formas de mejorarlo. En esta sección, nos dedicaremos a analizar y comparar los tiempos de ejecución de diferentes implementaciones para entradas de tamaños variados.

Comencemos explicando cuáles son las diferentes optimizaciones posibles del DSU. El *Disjoint Set Union* es la estructura de datos que le permite al algoritmo de Kruskal unir las diferentes componentes conexas de nuestro grafo en forma eficiente. Ésta cuenta con dos operaciones: **find()**, que dado un elemento devuelve el conjunto al que pertenece, y **unite()**, que dados dos conjuntos diferentes los une.

En particular, hay dos posibles optimizaciones para el DSU. El ya mencionado *Path Compression* es una mejora a la función de **find()**, donde en vez de recorrer varios elementos del conjunto hasta llegar al representante del mismo se devuelve el representante directamente, comprimiendo de esta forma el camino recorrido. Por otro lado, *Union by Rank* es una optimización de la función **unite()**, donde al unir dos conjuntos buscamos que el de menor cardinal se acople al de mayor, logrando que se modifiquen la menor cantidad de referencias posibles. Esto puede lograrse almacenando el tamaño de cada conjunto en un vector aparte y operando en él en tiempo constante.

Por otro lado, podemos comparar tiempos de ejecución con una implementación de Kruskal optimizada para grafos densos, donde si se tienen n vértices y m aristas, se da que $m \sim n^2$. Ésta versión del algoritmo no utiliza una estructura de DSU sino más bien una matriz de adyacencias, y no requiere de ordenar las aristas. Puede demostrarse que la complejidad de ésta implementación es $O(n^2)$, mejor al $O(n^2 * log(n^2)) = O(n^2 * log(n))$ de Kruskal con DSU. Por cómo funciona nuestra solución, el grafo que modela el problema de los modems es uno completo, con lo cual se esperaría que ésta implementación para grafos densos sea más rápida.

Para esta experimentación comparamos los tiempos de ejecución de las tres implementaciones del algoritmo de Kruskal (DSU sin y con optimizaciones, y para grafos densos) con casos de test random generados en Python. Se consideraron la cantidad de oficinas $N \in [100, 200, ..., 1000]$, y para cada N diferente se generaron 100 test al azar respetando las restricciones del enunciado original. De este modo, se tomó el promedio de los runtime para cada tamaño de entrada diferente y graficamos las curvas de cada algoritmo en el siguiente cuadro[3]:

Como se puede apreciar en el gráfico, la implementación optimizada para grafos densos es drásticamente más rápida, en promedio dos veces mejor que las otras dos (esto aumentaría a mayor tamaño de entrada). Ésto verifica lo previamente estimado y muestra que esta versión del algoritmo es la mejor elección de entre las tres para este problema particular. Por otro lado, podemos ver que no hay mucha diferencia en tiempos de ejecución entre ambas implementaciones con DSU. Mientras que claramente la implementación para grafos densos de Kruskal es la más rápida, utilizar un DSU básico u optimizado no hace diferencias notables en tiempos de ejecución. Esto podría deberse a que la complejidad dominante del algoritmo es el ordenamiento, muy superior a la de las operaciones del DSU. Luego, podría ser que a causa de la variación en la velocidad del ordenamiento (en nuestro caso QuickSort) no se pueden apreciar mejoras entre ambas versiones. Creemos que, en teoría, aumentando considerablemente la cantidad de tests cases por cada tamaño de entrada N diferente, eventualmente se reduciría esta variación, permitiendo ver una mejora en tiempos de ejecución del DSU optimizado.

[3]Los tests fueron ejecutados desde el mismo ordenador para todos los algoritmos, con un CPU Intel Core i5 6400 (4-core) y 16gb de RAM DDR4 (1066MHz).

3. Conclusiones

A lo largo de este informe, analizamos distintas alternativas para encarar el problema hasta obtener una solución correcta. Al hallar un algoritmo adecuado, experimentamos con casos de test aleatorios de diferentes tamaños y estudiamos su complejidad. A su vez, navegamos distintas implementaciones de la solución, buscando maximizar la eficiencia.

En conclusión, el problema de los módems puede resolverse correctamente realizando varias iteraciones del algoritmo de Kruskal, lo cual ya demostramos que halla una solución óptima.