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 $v \rightarrow w$ y $w \rightarrow v$, 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 $v \rightarrow w$ y $w \rightarrow v$, 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 $s \rightarrow v \rightarrow w \rightarrow t$ 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 $\rightarrow O(M + K)$
- Aplicar dijkstra $\rightarrow O(dijkstra)$
- Buscar camino mínimo con nuevos caminos $\rightarrow O(K)$

Como se puede apreciar, la complejidad total del algoritmo es O(m + k) + O(dijkstra) por lo que depende estrechamente del algoritmo de dijkstra que se utilice. A continuación presentaremos dos implementaciones diferentes.

La idea del algoritmo de dijkstra, en resumen, es a partir de un nodo inicial, llevar cuenta de a que nodos nuevos podemos acceder y tomar 'golosamente' el camino que nos permita acceder a un nodo desconocido de manera más barata. Luego a través de este nuevo nodo podremos acceder a otros, por lo que hay que actualizar nuestra estructura, incluyendo todas las nuevas opciones de nodos a las que podemos acceder sumándole la distancia que nos costó llegar al nodo padre. Y así repetir este ciclo hasta que no podamos acceder a ningún nodo nuevo.

Algoritmo 2: Pseudocódigo para dijkstra implementado con heap

```
struct arista {int nodo, peso}
proc dijkstra_heap(int v, vector<vector<arista>> grafo) : vector<int> {

    min_heap<arista> h
    h.push({v, 0})
    vector<bool> esta?(n, false)
    vector<int> dist(n, inf)
    dist[v] := 0

    while(not h.vacio?()):
        int w := h.minimo().nodo
        h.elimino_minimo()
        if(esta?[w]) continuo
        esta[w] = true
        for(arista u : grafo[w]) :
            if(dist[u.nodo] > dist[w] + u.peso) :
                dist[u.nodo] := dist[w] + u.peso
                h.push({u, dist[u]})

    return dist
}
```

En esta primera implementación la estructura con la que representamos los nodos accedibles en cada iteración del ciclo es un heap. Se puede observar que se hacen $d(v)$ inserciones al heap, para cada $v \in V$, que sabemos que en total suman $2*M$, y el while corre hasta que el heap se encuentre vacío, por lo que además haremos $2*M$ extracciones del heap. Ambas operaciones, inserción y extracción tienen una complejidad de $O(log(M))$. Podemos concluir entonces, que la complejidad del algoritmo es $O(M * log(M))$. Cabe aclarar que si el grafo es denso es decir $M \approx n^2$ entonces el algoritmo realizará $O(N^2 * log(N^2))$.

Algoritmo 3: Pseudocódigo para dijkstra implementado para grafos densos

```
struct arista {int nodo, peso}
proc dijkstra_heap(int v, vector<vector<arista>> grafo) : vector<int> {
    vector<bool> esta?(n, false)
    vector<int> dist(n, inf)
    dist[v] := 0
    for (int i : 0 .. n−1) :
        int mn := inf
        int w := −1
        for(int j : 0 .. n−1) :
            if(esta?[j]) continuo
            if(dist[j] < mn)
                mn := dist[j]
                w = j;

        if(w = −1) break

        esta?[w] = true;
        for(arista u : grafo[w])
            dist[u.nodo] := min(dist[u.nodo], dist[w] + u.peso)

    return dist;
}
```

En esta implementación la estructura con la que representamos los nodos accedibles en cada iteración del ciclo es un vector, donde se almacena el mínimo costo para acceder a cada nodo. Analizando la complejidad, notamos que por cada i de 0 a n-1, realizamos n operaciones y además realizamos $d(v)$ operaciones para cada $v \in V$, que como mencionamos previamente, la suma total es $2*M$. Es decir que la complejidad del algoritmo es $O(N^2+M)$ pero además sabemos que $M \leq N^2$, o sea que la complejidad del algoritmo es $O(N^2)$. Como observamos antes, esta implementación es más conveniente que la otra si el grafo es suficientemenete denso.

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.