DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2: Recorridos y Árbol generador mínimo

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

El *problema de los modems* consiste en calcular la mejor forma de brindarles acceso a internet a un conjunto de oficinas. Dadas N oficinas, se tienen W modems para instalar en ellas, con $W < N$. El problema surge de que la cantidad de modems es menor, con lo cual se deben de conectar con cables algunas oficinas entre sí para compartirse conexión. Hay dos tipos de cables: los UTP y los de fibra óptica, con precios por centímetro U y V respectivamente. La diferencia entre estos cables es que los UTP suelen ser más baratos, $U \leq V$, aunque solo se pueden utilizar para conectar dos oficinas que estén a distancia R o menor.

Entonces, dadas las posiciones de las N oficinas en un eje cartesiano medido en centímetros, la cantidad W de modems, los precios U y V de ambos cables y la distancia R, debemos calcular el mínimo costo en cables necesario para conectar a todas las oficinas. Además, se tiene que aclarar cuánto se gasta en cables UTP y de fibra óptica, respectivamente[1].

2. Resolución

Tenemos W módems y queremos que con ellos todas las oficinas dispongan de conexión, minimizando a la vez el costo en cables. Primero, presentaremos una posible solución al problema explicada paso a paso, cuya correctitud será demostrada más adelante. El *problema de los módems* puede ser resuelto con el siguiente algoritmo:

1. Armamos un grafo completo donde cada oficina es su propio vertice, y cada una tiene una arista con todas las demás. El peso de cada arista es la distancia en centímetros entre ambas oficinas.
2. Ordenamos las aristas por peso.
3. Aplicamos el algoritmo de Kruskal hasta obtener N-W aristas, o mismo, un bosque con W componentes conexas. Luego, en cada una de estas componentes asignamos un módem.
4. Por cada arista agregada a la solución, sumamos a la respuesta final su peso multiplicado por el costo del cable (eligiendo UTP como material siempre que sea posible).
5. Retornamos la suma de los costos en cables UTP y fibra óptica.

La idea detrás del mismo es agrupar a todas las oficinas en W conjuntos disjuntos, los cuales dispondrán de un módem cada uno. De esta forma, cada oficina tendrá acceso a exáctamente un módem. Puede demostrarse que en una solución minimal, ninguna oficina puede estár conectada a más de uno. Luego, aplicando el algoritmo de Kruskal, se pueden hallar las W componentes conexas tal que minimicen los costos de los cables y así obtener la solución al problema.

2.1. Demostración de Correctitud

Primero vamos a observar que en una solución minimal no puede haber más de W árboles en el bosque. Esto se debe a que, como tenemos W módems, no tendríamos un módem por árbol, por lo que habría al menos un árbol de oficinas que quedaría sin conexión.

Luego queremos probar que en una solución que minimiza los costos de los cables no puede haber menos de W árboles. Para lograrlo, asumimos que existe una solución mínima tal que se cumple que las oficinas están repartidas en menos de W conjuntos conexos. Podríamos entonces

[1] Restricciones a los parámetros:
$1 \leq W < N \leq 1000 \ \wedge \ 1 \leq U \leq V \leq 10 \ \wedge \ 1 \leq R \leq 10000 \ \wedge \ -10000 \leq x_i, y_i \leq 10000$.

darle un módem a alguna oficina de cada árbol, de esta manera todas las oficinas tendrían conexión. Como tenemos menos de W árboles, nos sobrarían módems, entonces, si elegimos una oficina "hoja", es decir que tenga una sola arista, y le asignamos uno de los módems que nos sobraron, estaríamos recortando costos ya que podríamos eliminar la conexión que esta oficina tenía previamente. Por lo que llegamos a un absurdo, concluyendo que la asumpción de que teníamos una solución minimal es falsa.

De esta manera probamos que en una solución minimal la cantidad de árboles no es mayor ni menor a W, por lo que notamos que debe ser exactamente W.

Encontrar las W componentes conexas tal que minimicen los costos se puede lograr siguiendo el invariante del algoritmo de Kruskal. Esto es porque luego de i pasos de Kruskal, tendremos $N - i$ componentes conexas, tal que no existe otro bosque de i aristas que sume a un costo menor. Entonces, luego de realizar $N - W$ pasos, tendremos $N - (N - W) = W$ conjuntos disjuntos de oficinas, conectadas entre sí minimizando los costos.

Queda entonces demostrado que el algoritmo planteado es correcto ya que demostramos que para hallar una solución que minimice los costos es necesario dividir a las oficinas en exactamente W componentes conexas y aprovechamos el algoritmo de Kruskal que nos ofrece una manera de encontrar dicha división.

2.2. Implementación

Presentamos a continuación una posible implmentación de la solución explicada:

Algoritmo 1: Pseudocódigo

```
struct arista {int u, v, d}
proc modems(in N, R, W, U, V: int, in vector<coordenada> pos) {
    // pos tiene las coordenadas de todas las oficinas
    // Generamos y ordenamos aristas por peso
    vector<arista> aristas := {}
    for (i := 0; i < N; i := i + 1) :
        for (j := i+1; j < N; j := j + 1) :
            peso := sqrt((pos[i].x - pos[j].x)^2 +
                         (pos[i].y - pos[j].y)^2)
            aristas.push_back({i,j,d})

    sort(aristas) // QuickSort
    // Realizamos N - W iteraciones de Kruskal
    dsu grupos(N) := {1...N}
    pU := 0
    pV := 0
    faltan := N - W
    for (arista e: aristas) :
        if (grupos.get(a.u) != grupos.get(a.v)) :
                grupos.unite(a.u, a.v);
                if (a.d ≤ R) : pU += a.d;
                else :         pV += a.d;
                faltan := faltan - 1
                if (faltan = 0) : break

    res := {pU * U, pV * V}
}
```

2.3. Análisis de la Complejidad

La complejidad del algoritmo presentado es la siguiente:

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

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.

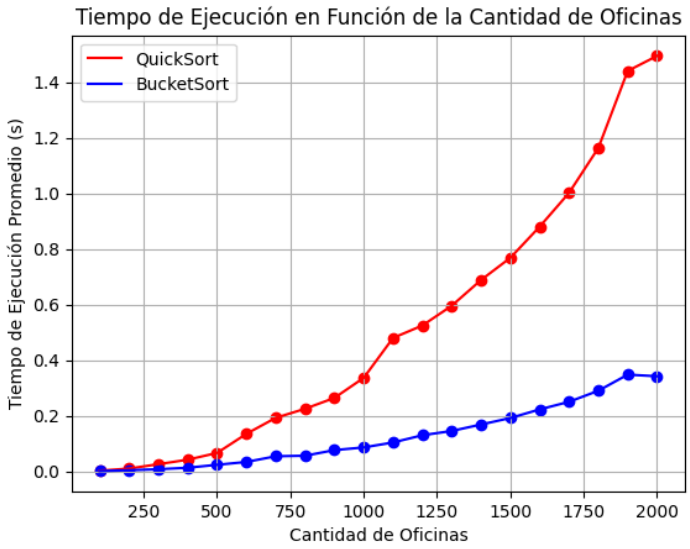
Agrupando las aristas de esta manera, sabemos que para toda arista que se halla en el i-ésimo bucket, su peso será menor al de cualquiera de los buckets mayores a i.

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]:

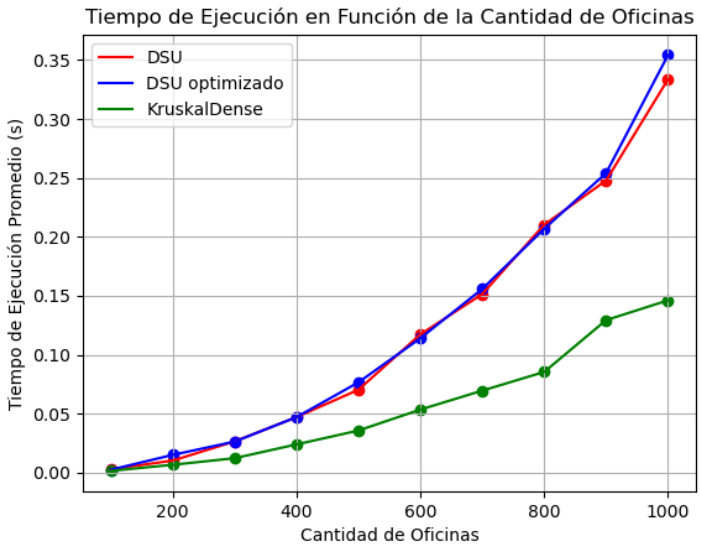


Figura 1: Tiempos de ejecución de las distintas implementaciones de Kruskal con entradas random.

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. 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.

[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).