DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Recorridos y árbol generador mínimo

28 de Mayo de 2023 — Algoritmos y Estructuras de datos III

Integrante	LU	Correo electrónico
Andrés Finkelsteyn	511/01	andresgfin@gmail.com
Martín Santesteban	397/20	martin.p.santesteban@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

1. Descripción del problema

Se quiere proveer de internet a N oficinas del subsuelo del pabellón $0 - \infty$. Para ello, se compraron W módems, los cuales pueden instalarse en cualquier oficina, brindándole acceso a internet. Como $W < N$ se deberán comprar también cables para conectar algunas oficinas entre si. Se pueden comprar cables UTP o de fibra óptica, los cuales tienen un costo por metro de U y V, respectivamente. Los cables UTP tienen la condición particular de que sólo pueden usarse entre dos oficinas si estas están a menos de R centímetros. Dadas las posiciones de las oficinas en un eje cartesiano (expresadas en centímetros), la cantidad de módems adquiridos, el precio por metro de los cables UTP y de fibra óptica, y la distancia R, debemos encontrar cuál es la mínima cantidad de dinero que debe pagarse en cables para conectar todas las oficinas. Puntualmente, queremos saber cuánto debe gastarse en cables UTP y cuánto en cables de fibra óptica.

2. Descripción del algoritmo

El algoritmo consiste en tres partes: en la primera se define un grafo pesado, en la segunda se busca para dicho grafo un bosque generador mínimo de W componentes conexas y en la tercera se calculan los costos de los cables en base a los pesos del bosque generador mínimo.

En el grafo pesado los nodos son las oficinas y las aristas unen todos los pares de oficinas entre sí con un peso igual a la distancia entre ellas. Se puede notar que se trata de un grafo pesado completo de N nodos.

El bosque generador mínimo es hallado mediante el algoritmo de Kruskal, aprovechando el invariante según el cual luego del agregado de la i-ésima arista se obtiene un bosque de N -i componentes conexas. Por lo tanto el algoritmo de Kruskal se detiene luego de agregar N - W aristas, de modo que se obtiene un bosque generador de W componentes conexas.

El cálculo de la respuesta al problema se realiza teniendo en cuenta la relación entre U y V. Si $V \leq U$ se calcula el costo de cada arista como el producto de V y su peso. Si $V > U$ el costo de una arista se calcula como el producto de U y su peso si dicho peso es menor o igual que R, en el caso contrario el costo se calcula como el producto entre V y su peso. Los costos de las aristas representan los costos de los cables, por eso se usan para calcular las sumas de costos parciales. La primer suma incluye a los cables cuyo costo por metro es U y la segunda suma incluye a los cables cuyo costo por metro es V.

3. Justificación de correctitud

Si se modela el problema como un grafo en el cual los nodos son las oficinas y las aristas tienen pesos dados por las distancias entre sus correspondientes oficinas, este se reduce a cumplir los siguientes objetivos:

- Búsqueda por el algoritmo de Kruskal de un bosque generador mínimo de W componentes unidas por aristas cuyo peso viene dado por la distancia entre los dos vértices.
- Asignación a cada arista del menor costo posible eligiendo entre las dos opciones de costo por metro. Los costos asignados corresponden a los costos de los cables del problema.
- Cálculo de las sumas de los costos parciales. Por un lado se calcula la suma correspondiente a los cables cuyo costo por metro es U y por otro lado la suma de los cables cuyo costo por metro es V.

Se explicará por qué cumplir estos objetivos resuelve el problema. Para ello se debe notar que la existencia de W componentes conexas garantiza que en cada una de ellas se puede situar un módem para proveer de internet a todas sus oficinas. Si el número de componentes fuera mayor que W las oficinas de al menos una componente se quedaría sin internet y si el número fuera menor se usarían más cables, por lo cual el costo no sería mínimo. Como resultado de aplicar el algoritmo de Kruskal se sabe que se encuentra el bosque en el cual las aristas son las de menor peso que se pueden seleccionar dentro del conjunto de todas las aristas. Por lo tanto si en el cálculo del costo de cada arista se elige el menor valor de costo por metro entre los cables disponibles se garantiza que cada arista tenga el menor costo posible también. Para usar ese criterio en la elección hay que considerar si la arista tiene un peso menor que R. En el caso afirmativo, si $V \leq U$ el costo de la arista debe ser el producto entre V y su peso y si $V > U$ debe ser el producto entre U y su peso. En el caso contrario, el costo de la arista se calcula como el producto entre V y su peso. De esa forma el costo total resulta ser el mínimo posible y la respuesta al problema se reduce a devolver las sumas de los costos correspondientes a cada tipo de cable como indica el tercer objetivo. El algoritmo presentado previamente completa los objetivos mencionados por lo tanto es correcto.

4. Experimentación

Esta sección consiste en el estudio empírico de los tiempos de ejecución del algoritmo de Kruskal considerando distintas implementaciones. Por un lado se comparan los tiempos de las implementaciones del algoritmo de Kruskal diseñadas para grafos densos y ralos. Por otro lado se comparan dentro del algoritmo de Kruskal tradicional los tiempos de las optimizaciones sobre la estructura de datos empleada para los conjuntos disjuntos.

4.1. Experimentación sobre las distintas implementaciones del algoritmo de Kruskal.

El objetivo de este experimento es comparar los tiempos de ejecución de dos implementaciones distintas del algoritmo de Kruskal en función de la cantidad de oficinas.

La primera implementación es la más tradicional del algoritmo de Kruskal y también la más apropiada para grafos ralos. El ordenamiento de las aristas de acuerdo a su peso se realizó con la función $std :: sort()$ de la librería estándar de $C++$, con complejidad $O(n * log(n))$. La estructura de datos que fue utilizada para implementar los conjuntos disjuntos consta de un conjunto de árboles (uno por componente conexa) donde cada árbol es representado por la raíz del mismo. Para mantener la estructura de datos solo se requieren dos vectores, uno que mantiene las relaciones de parentesco entre los nodos y otro que contiene en la i-ésima posición la altura del árbol con representante i.

Nos interesa poder unir componentes conexas, y dado un nodo poder determinar a qué componente pertenece. Para saber a qué componente pertenece un nodo basta con determinar cuál es la raíz del árbol, operación conocida como $Find$. Esta función tiene una estructura recursiva, donde se avanza nivel por nivel en el árbol hasta llegar a la raíz. Para la unión de componentes conexas usamos $UnionByRank$: al unir un árbol de raíz u y otro de raíz v, en caso de que el primero tenga mayor altura que el segundo se asigna a u como padre de v produciendo un árbol en el que la componente de v es un subárbol hijo de u. Ésta Heurística la utilizamos para mantener un árbol equilibrado, haciendo que se recorra una cantidad de nodos en el orden logarítmico durante la ejecución de $Find$. La unión tiene complejidad constante, ya que basta ejecutar una asignación en el vector de padres.

Sabemos que se ejecutaran $n - 1$ uniones entre componentes y como mucho $2m$ operaciones $Find$, por lo que la complejidad teórica final termina siendo $O(m.log(n) + (n - 1) + m.log(n)) = O(m.log(n) + n)$. Como estamos trabajando con grafos completos, para el problema en cuestión $m \in O(n^2)$ y como consecuencia la complejidad final es $O(n^2.log(n))$.

La segunda implementación está pensada para grafos densos y adopta una estrategia diferente respecto de las implementaciones anteriores. El bosque es modelado con un grafo que se representa con una matriz de adyacencias, lo cual se puede hacer porque la declaración de dicha matriz no afecta la complejidad total. Cada componente conexa es representada por un único nodo r en la matriz de adyacencia que mantiene las aristas de mínimo peso incidentes a otra componente conexa. Es decir, al unir una componente conexa representada por u con otra componente representada por v en la iteración $j+1$ tenemos que $\forall i \in V$ si i es el representante de una componente conexa entonces $M_{j+1}[u][i] = M_{j+1}[i][u] = min\{M_j[u][i], M_j[v][i]\}$.

La estrategia para unir las componentes se mantiene igual, se realizan n iteraciones uniendo componentes con la arista de menor peso que no genera ciclos. Sin embargo, a diferencia de las demás implementaciones, no se ordenan las aristas sino que se mantiene un vector que almacena en la i-ésima posición la arista de menor peso que incide en el representante i y al mismo tiempo incide en otra componente. Tanto la función utilizada para encontrar la arista de menor peso como la utilizada para unir dos componentes tienen complejidad $O(n)$ y dado que se realizan n iteraciones esto resulta en una complejidad total de $O(n^2)$.

4.1.1. Hipótesis

Al medir los tiempos de ejecución en función de la cantidad de oficinas, se espera ver que la versión pensada para grafos densos sea considerablemente más rápida que la primer implementación, debido a que se trabaja con grafos completos y por lo tanto solo ordenar las aristas tiene una complejidad teórica mayor a $O(n^2)$.

4.1.2. Procedimiento

Se generaron 100 instancias de forma aleatoria para cada tamaño de instancia donde los tamaños fueron de la forma $N \in \{50, 150, 200, ..., 2000\} \cup \{3000, 4000, ..., 10000\}$. A partir de los datos obtenidos se graficaron los tiempos de ejecución de cada método, junto a una regresión lineal de los datos transformados para intentar demostrar que las complejidades teóricas se verifican en la práctica.

4.1.3. Resultados

Se puede observar en la figura 1 que efectivamente la implementación para grafos densos tiene mejores tiempos de ejecución que el algoritmo de Kruskal tradicional.

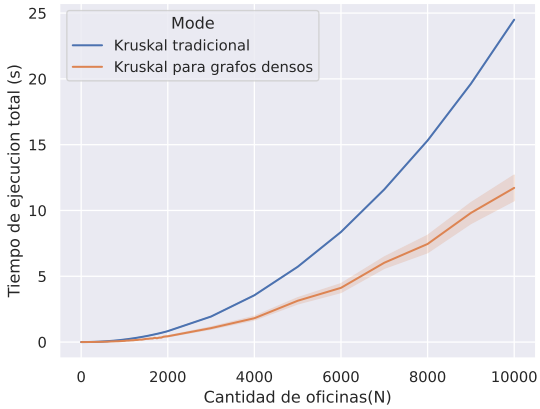
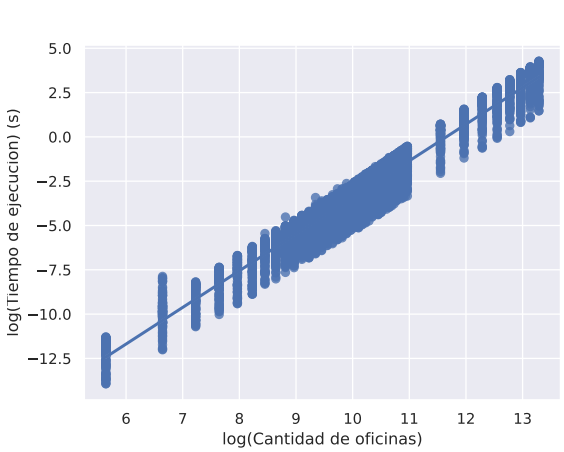
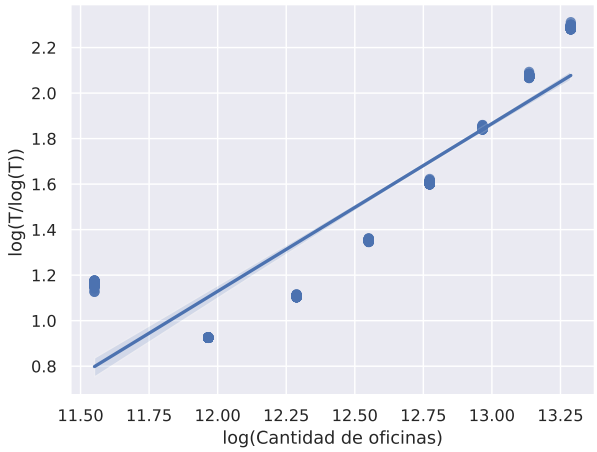


Figura 1: Tiempo total de ejecución *vs* Cantidad de oficinas.

Con el fin de estudiar si el algoritmo de Kruskal para grafos densos tiene complejidad cuadrática se analizó la relación lineal entre la cantidad de oficinas y los tiempos de ejecución ambos transformados por la función logarítmica. La figura 2a muestra que efectivamente la versión para grafos densos tiene complejidad cuadrática porque la transformación logarítmica resulta en una recta con un coeficiente de correlación igual a 0.976 y una pendiente de 2,075.



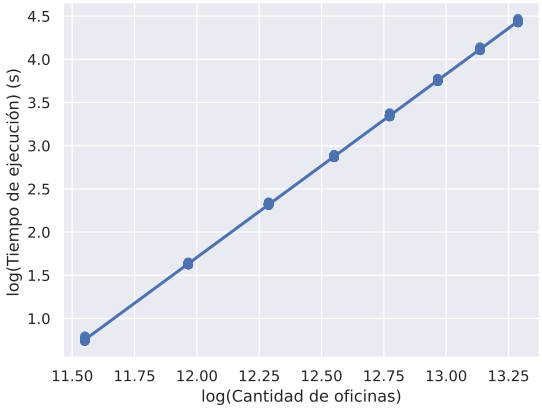
(a) Regresión lineal en escala logarítmica de Kruskal para grafos densos.

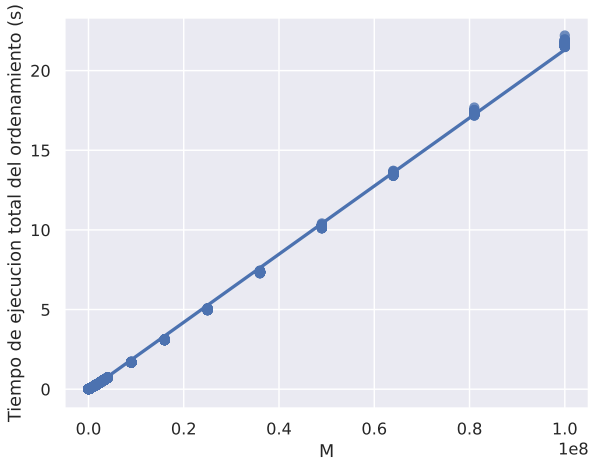


(b) Regresión lineal en escala logarítmica de Kruskal para grafos ralos, donde T es el tiempo de ejecución.

Con el fin de determinar si la complejidad de la versión de Kruskal para grafos ralos es $O(n^2.log(n))$ se dividieron los tiempos de ejecución por el logaritmo de n y luego se realizó la transformación logarítmica de las variables, los datos resultantes se exponen en la figura 2b. El coeficiente de correlación es 0,9505 pero sorpresivamente la pendiente es 0,734 cuando se esperaba un número cercano a 2, indicando que en la práctica los tiempos no se ajustan con la complejidad teórica. Con esto en mente, se volvieron a analizar los datos en forma similar con la diferencia de que no se dividieron los tiempos de ejecución por el logaritmo de n. Los resultados que se encuentran exhibidos en la figura 3a dieron una pendiente de la recta igual a 2.12 y un coeficiente de correlación de 0,999. En consecuencia los datos parecen indicar que los tiempos de ejecución tienen orden cuadrático y no $O(n^2.log(n))$.

Luego de analizar estos resultados se advirtió que los tiempos de ejecución posteriores al ordenamiento son despreciables en comparación al tiempo que requiere la función *std* :: *sort*. Se decidió estudiar el tiempo de ejecución de *std* :: *sort* en función del número de aristas y se descubrió que la relación es lineal, como muestra la figura 3b. Aparentemente esto se debe a las numerosas optimizaciones que tiene el algoritmo de ordenamiento estándar de $C++$, ya que la complejidad esperada en teoría es $O(m.log(m)) = O(n^2.log(n))$. De esta forma se pudo explicar por qué los resultados no se ajustaron a la complejidad teórica y presentaron un orden de complejidad cuadrático.





(a) Regresión lineal en escala logarítmica de Kruskal para grafos ralos.

(b) Regresión lineal entre los tiempos de ejecución de *std* :: *sort* y M (número de aristas).

4.2. Experimentación sobre las optimizaciones de DSU.

El objetivo de este experimento es medir los tiempos de las distintas optimizaciones de *DSU* que usa el algoritmo de Kruskal tradicional para grafos ralos. Se busca experimentar con heurísticas para la unión de componentes *UnionByRank* y *UnionBySize*, que se usan para mantener el árbol de cada componente conexa equilibrado. Esto permite de optimizar la ejecución de la búsqueda del representante de la componente conexa a la que pertenece un nodo. *UnionByRank* une dos árboles en función a la altura de cada uno, mientras que *UnionBySize* los une en función a la cantidad de nodos que forman parte de la componente: el árbol con menos nodos pasa a ser subárbol de la componente con más nodos. *PathCompression* es una optimización que se usa en la operación *Find*: luego de consultar cuál es el representante de un nodo, todos los nodos del camino recorrido hasta la raíz pasan a ser hijos directos de la raíz. De ésta forma, todas las consultas a futuro sobre el nodo o cualquiera en el camino simple hasta la raíz pasan a tener tiempo prácticamente constante. Es importante mencionar que adicionalmente se van a tomar los tiempos para la versión *Null* que no cuenta con una heurística para unir componentes ni con *PathCompression*. Al unir dos componentes u y v siempre decide que el árbol de v será subárbol de u.

4.2.1. Hipótesis

Se espera ver que todas las versiones tienen mejores tiempos de ejecución que *Null* debido a que ésta versión no solo no cuenta con optimizaciones sino que además produce árboles no balanceados. Esto se debe a que si u siempre está constituido por un único nodo, al mantenerlo como raíz de la unión se genera un árbol con una única hoja y forma de rama.

Con respecto a las demás versiones, se espera ver que *UnionByRank* y *UnionBySize* tengan tiempos de ejecución muy similares, debido a que mantener árboles balanceados deja que *Find* tenga complejidad $O(log(n))$. Por último, *PathCompression* debería traer mejoras con respecto a los tiempos de ejecución ya que *Find* tiene complejidad $O(\alpha(n))$ donde α es la inversa de Ackerman y $O(\alpha(n)) \subset O(log(n))$.

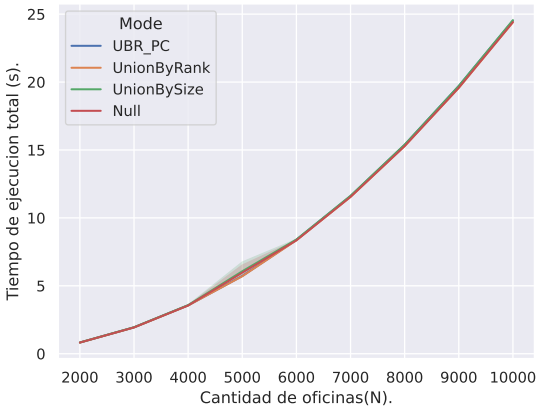
4.2.2. Procedimiento

Se generaron instancias de forma aleatoria para $N \in \{3000, 4000, ..., 10000\}$ y se tomaron los tiempos de ejecución totales y parciales, posteriores al ordenamiento de las aristas. Se graficaron los promedios para cada tamaño junto con la desviación estándar de las mediciones (franjas de colores mas claros).

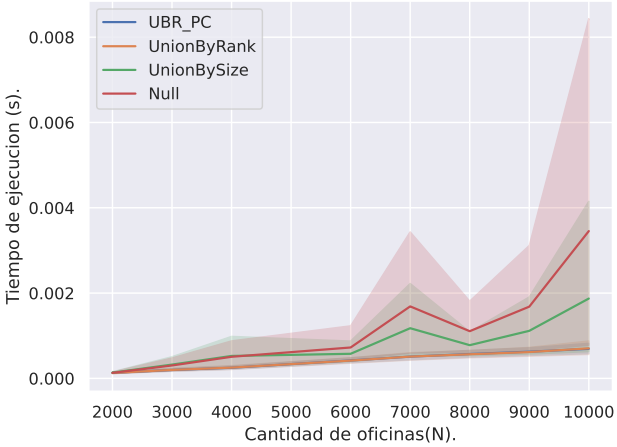
4.2.3. Resultados

La figura 4a expone los tiempos de ejecución totales para todas las versiones. Sorprendentemente las optimizaciones no parecen presentar diferencias en los tiempos de ejecución, sin embargo resulta que los tiempos de ejecución de las operaciones sobre la estructura de *DSU* son despreciables en comparación al tiempo del ordenamiento. Con esto en mente, el gráfico de la figura 4b expone los tiempos de ejecución luego del ordenamiento para cada versión. Se puede observar que *Null* es el que más tarda, seguido por *UnionBySize*, y por último *UnionByRank* ya sea con *PathCompression* o sin dicha función. La diferencia en tiempos de ejecución es mucho menor de lo que se esperaba, sin embargo es importante notar que la desviación estándar de *Null* es muy grande, exponiendo la existencia de casos en los que no tener una heurística para mantener los árboles equilibrados implica que *Find* ejecute más operaciones. Lo mismo sucede con *UnionBySize* pero en menor escala. La versiones de *UnionByRank* con *PathCompression* y

sin dicha función mostraron ser más eficientes que el resto de las optimizaciones, sin embargo, como expone la figura 5, *PathCompression* no trajo ningún tipo de mejora.



(a) Tiempo total de ejecución *vs* Cantidad de oficinas.



(b) Tiempo de ejecución posterior al ordenamiento *vs* Cantidad de oficinas.

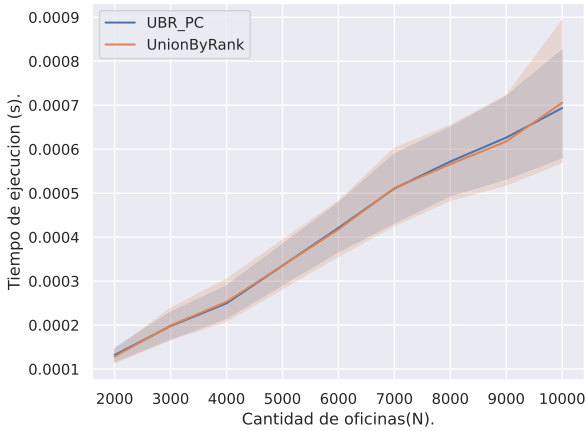


Figura 5: Tiempo de ejecución posterior al ordenamiento *vs* Cantidad de oficinas.