Aplicación de Ordenamiento en Línea: Construcción Eficiente del Árbol Cobertor Mínimo *

Rodrigo Paredes
Centro de Investigación de la Web,
Universidad de Chile, Depto. de Ciencias de la Computación,
Santiago, Chile, Blanco Encalada 2120.
raparede@dcc.uchile.cl

Abstract

An *on-line sorting* application is presented: the optimization of the Kruskal's algorithm for Minimum Spanning Tree construction of a given graph $G(V, E)$. This optimization allows one to reduce Kruskal's algorithm temporal complexity from $O(|E| \log |E|)$ to an empirical time cost of the form $C_1 \cdot |E| + C_2 \cdot k \log k$, where k is the size of the edge subset of G that is reviewed during the minimum spanning tree construction. In practice, k is much less than $|E|$, and in the case of complete graphs, this optimization has better performance than Prim's algorithm.

Keywords: Kruskal's algorithm, Graphs, On-line sorting.

Resumen

Se presenta una aplicación del *ordenamiento en línea*: la optimización del algoritmo de Kruskal para construir el Árbol Cobertor Mínimo de un grafo $G(V, E)$ dado. Esta optimización permite reducir la complejidad temporal del algoritmo de Kruskal de $O(|E| \log |E|)$ a un costo empírico de tiempo de la forma $C_1 \cdot |E| + C_2 \cdot k \log k$, donde k es el tamaño del subconjunto de aristas de G que se revisan al construir el árbol cobertor mínimo. En la práctica k es mucho menor que $|E|$, y en el caso de grafos completos, esta optimización tiene mejor desempeño que el algoritmo de Prim.

Palabras claves: Algoritmo de Kruskal, Grafos, Ordenamiento en línea.

*Este trabajo ha sido financiado por el Núcleo Milenio Centro de Investigación de la Web, Proyecto P01-029-F, Mideplan, Chile.

1. Introducción

Sea $G(V, E)$ un grafo conexo no dirigido con pesos no negativos asignados a sus aristas $e \in E$. Un Árbol Cobertor Mínimo (ACM, o MST del inglés Minimum Spanning Tree) de un grafo G es un *árbol* porque es acíclico, es *cobertor* porque conecta todos los vértices y es *mínimo* porque la suma de los costos de las aristas del ACM es la menor posible. Note que dado un grafo $G(V, E)$, el ACM puede no ser único, el número de aristas del ACM es $|V| - 1$ y para construir el ACM sólo se pueden utilizar las aristas de G.

Los álgoritmos básicos para construir el ACM son de tipo *greedy* (algoritmos ávidos). Aunque en general esta estrategia no garantiza que se encuentre un óptimo global, pues es un método que sólo optimiza las decisiones de corto plazo, se puede demostrar que para este problema siempre logra alcanzar el óptimo.

La idea básica greedy consiste en construir el árbol cobertor mínimo utilizando los árcos de menor costo que eviten la creación de un ciclo. El costo del árbol resultante no se puede mejorar (es decir, bajar) puesto que si se reemplaza cualquier arco del árbol por otro del grafo que no esté en el árbol, necesariamente el costo del árbol resultante tras la modificación será igual o mayor.

Las técnicas más conocidas para construir el ACM son el algoritmo de *Kruskal* [4] y el de *Prim* [6], que tienen complejidades temporales $O(|E| \log |E|)$ y $O(|V|^2)$, respectivamente. Para grafos ralos, es decir, de densidad de arcos baja, $|E| \in O(|V|)$, se recomienda utilizar el algoritmo de Kruskal y para grafos densos, es decir, de densidad de arcos alta, $|E| \in \Theta(|V|^2)$, el de Prim [2, 8]. Existen otros algoritmos para resolver este problema recopilados por Tarjan en [7] y más recientes en [1], en este último, el autor muestra un algoritmo que toma tiempo $O(|E|\alpha(|E|, |V|))$, donde α es la inversa de la función de Ackermann que es una función de crecimiento muy lento por lo que el costo del algoritmo se aproxima al límite teórico $O(|E|)$, este algoritmo es bastante complejo, luego su interés es principalmente teórico.

Estudios experimentales se tienen en [5, 3]. En [5] se estudian varias versiones de los algoritmos de Kruskal, Prim y Tarjan. En [3] se estudia un algoritmo basado en la propiedad del ciclo: el arco más pesado de un ciclo no pertenece al ACM.

La complejidad temporal del algoritmo de Kruskal está dominada por el tiempo que toma la ordenación del conjunto de aristas E, que es $O(|E| \log |E|)$. En este trabajo se presenta una optimización del algoritmo de Kruskal basado en un algoritmo de *ordenamiento en línea* (OLQS del inglés On-Line QuickSort) del conjunto de aristas E, en donde la ordenación del conjunto de aristas se limita sólo a la porción de aristas necesarias para construir el ACM. Con esta optimización, y considerando que la porción a ordenar es de tamaño k, se consigue un costo empírico de tiempo de la forma $C_1 \cdot |E| + C_2 \cdot k \log k$, el cual en práctica es 13 veces más rápido que el tiempo de CPU que toma el algoritmo de Kruskal, para grafos de densidad de arcos media o alta. Se destaca que para grafos completos, esta optimización del algoritmo de Kruskal tiene un mejor desempeño que el algoritmo de Prim.

La organización de este artículo es la siguiente: en la Sección 2 se muestra el algoritmo de Kruskal y la optimización utilizando heaps; en la Sección 3 se introduce el concepto de ordenamiento en línea; en la Sección 4 se muestra el algoritmo de Kruskal optimizado con ordenación en línea; los resultados experimentales se muestran en la Sección 5; por último, en la Sección 6 se muestran las conclusiones de este trabajo.

2. Algoritmo de Kruskal

La idea básica del algoritmo de Kruskal es comenzar con un bosque de $n = |V|$ árboles o componentes conexas, y luego fusionarlos hasta que se forme una única componente conexa. Para esto el algoritmo de Kruskal comienza con un subgrafo que contiene sólo los vértices del grafo original, sin aristas, es decir, cada vértice constituye su propia componente conexa. Luego, en cada iteración, se agrega al subgrafo el arco más barato del grafo original que no introduzca un ciclo, es decir, sólo se agregan aristas cuyos vértices pertenezcan a componentes conexas distintas. Una vez agregada la arista ambas componentes se fusionan en una sola. El proceso termina cuando el subgrafo constituye una única componente conexa.

Para administrar las operaciones con componentes conexas se utiliza la estructura de datos *UnionFind* con la heurística de compresión de caminos. UnionFind permite, dado un vértice u, determinar eficientemente cuál es la componente conexa a la que pertenece (*find(u)*), y dados dos vértices u y v fusionar sus componentes (*union(u,v)*). El costo amortizado de las operaciones find es $O(\log^* n)$ lo cual es casi constante, y el costo de las operaciones union es constante [2].

El algoritmo de Kruskal se muestra en la Figura 1. La inicialización de UnionFind toma tiempo $O(n)$, con $n = |V|$. La ordenación de las aristas del grafo toma tiempo $O(m \log m)$, con $m = |E|$. Se realizan a lo más m iteraciones del ciclo while, y cada iteración tiene costo constante. Luego, el tiempo de CPU de este algoritmo está dominado por el costo de ordenar las aristas según su peso, siendo la complejidad temporal del algoritmo de Kruskal de $O(m \log m)$.

```
Kruskal1 (Grafo G(V, E))

  UnionFind C ← {{v}|v ∈ V} // el conjunto de las componentes conexas
  ACM ← ∅ // árbol cobertor mínimo
  Ordenar las aristas de E en orden creciente de peso
  while |C| > 1 do
    Sea e = {v, w} la siguiente arista en orden creciente de peso
    if C.find(v) ≠ C.find(w) then
      ACM ← ACM ∪ {e}
      C.union(v, w)
  return ACM
```

Figura 1: Algoritmo de Kruskal básico.

2.1. Optimización del Algoritmo de Kruskal utilizando Heaps

Al analizar el funcionamiento del algoritmo de Kruskal se observa que para construir el ACM no es necesario revisar todas las aristas del grafo, sino que basta con ordenar un subconjunto de ellas que corresponden a las aristas más livianas del grafo. En el peor caso se necesitan revisar todas las aristas, lo cual, al igual que el algoritmo básico de Kruskal, ordena al conjunto de aristas completo.

La optimización utilizando heaps consiste en tomar el conjunto de aristas, transformarlo en un heap y luego extraer la arista de menor costo del heap en la medida que sea necesario, reemplazando de este modo la ordenación inicial de las aristas del grafo. El algoritmo de Kruskal optimizado con heaps se muestra en la Figura 2.

La inicialización de UnionFind toma tiempo $O(n)$. La construcción del heap toma tiempo $O(m)$. Se realizan a lo más m iteraciones del ciclo while, sea k la cantidad de iteraciones realizadas ($k \leq m$).

```
Kruskal2 (Grafo G(V, E))

  UnionFind C ← {{v}|v ∈ V} // el conjunto de las componentes conexas
  ACM ← ∅ // árbol cobertor mínimo
  heapify(E) // construye el heap con las aristas ordenado por peso
  while |C| > 1 do
    (e = {v, w}) ← extractMin(E) // la siguiente arista en orden creciente de peso
    if C.find(v) ≠ C.find(w) then
      ACM ← ACM ∪ {e}
      C.union(v, w)
  return ACM
```

Figura 2: Algoritmo de Kruskal optimizado con heaps.

En cada iteración se extrae el mínimo del heap, lo que cuesta $O(\log m)$, y las operaciones de union y find, que tienen costo constante. Luego, el tiempo esperado de este algoritmo es $C_1 \cdot m + C_2 \cdot k \log m$, y la complejidad de peor caso ($k = m$) es $O(m \log m)$.

3. Ordenamiento en Línea

Ordenamiento en línea se refiere a ordenar los primeros elementos de un conjunto bajo las siguientes condiciones:

1. Se conoce el conjunto de elementos a ordenar.
2. No se conoce a priori cuantos elementos del conjunto se quieren ordenar.
3. El tiempo de CPU esperado para ordenar los primeros elementos, sea el mismo tiempo del que se necesitaría de conocer cuántos elementos se van a ordenar.

La versión fuera de línea de este problema consiste en ordenar los primeros k elementos de un conjunto de tamaño m. Una forma de realizar esto es utilizar *QuickSelect* para encontrar el k-ésimo elemento del conjunto original, y luego ordenar el subconjunto de elementos menores al k-ésimo elemento. El tiempo esperado de QuickSelect es $O(m)$ y el tiempo de ordenar el subconjunto de elementos menores es $O(k \log k)$, con lo cual el tiempo total es de $O(m + k \log k)$.

Ordenar en línea los primeros elementos de un conjunto de tamaño m es equivalente a seleccionar en orden ascendente el primer elemento, luego el segundo y continuar hasta que arbitrariamente se detenga el proceso de selección de menores en algún valor k desconocido en $[1, m]$. Una forma de hacer esto es invocar k veces a QuickSelect para encontrar los primeros k elementos, con lo que el costo esperado de la ordenación en línea sería de $O(k \cdot m)$ (cada invocación de QuickSelect tiene costo esperado $O(m)$), lo que es superior al tiempo $O(m + k \log k)$ requerido.

El problema de esta estrategia es que no reutiliza el trabajo realizado en las invocaciones anteriores de QuickSelect. Por ejemplo, no se considera que cuando se selecciona el i-ésimo elemento, ya se han seleccionado $i - 1$ previamente. Por otro lado, cuando se realiza una llamada a QuickSelect se transforma el conjunto en un arreglo, que luego se particiona dos subarreglos: los elementos menores y mayores que el pivote, ubicando al pivote en la posición que le corresponde dentro del arreglo. Cuando se realice la siguiente llamada a QuickSelect sólo será necesario buscar el i-ésimo elemento en el subarreglo limitado por la posición i y la posición del último pivote.

Para almacenar las posiciones de los pivotes que se utilizan en los particionamientos, se utiliza p, que corresponde a una instancia de la estructura de datos *Pila*. Las operaciones de la pila son: *p.top()*, que retorna el elemento al tope de la pila; *p.push(x)*, que inserta al elemento x en el tope de la pila; y *p.pop()*, que extrae al elemento que está en el tope de la pila.

En la Figura 3 se muestra el algoritmo se selección incremental de elementos, que llamaremos *IQS*, del inglés *Incremental QuickSelect*, el cual deja en la posición *ini* del arreglo S al menor elemento del subarreglo $S[ini..p.\text{top}()]$.

```
IQS (Arreglo S, int ini, Pila p)

  if ini = p.top() then
    p.pop()
    return // el último pivote es el menor del subarreglo

  Seleccionar pivote piv al azar entre S[ini..p.top()-1]
  (S<, S>) ← Particionar(S[ini..p.top()-1], piv)
  S[ini..p.top()-1] ← [S<, piv, S>]

  if ini = piv then return // el pivote es el menor del subarreglo
  else p.push(posición(piv)) // guardando la posición del pivote

  // llamada recursiva en el subarreglo izquierdo
  IQS(S, ini, p)
```

Figura 3: Algoritmo de selección incremental de menores, note que el rango de la búsqueda se limita a $[ini \ldots p.\text{top}()]$.

3.1. Análisis simplificado de IQS

Para ordenar parcialmente hasta el k-ésimo elemento de un arreglo E, se busca al menor del arreglo y luego se seleccionan elementos menores incrementalmente hasta llegar al k-ésimo. Para calcular el trabajo total de esta operación, basta con sumar todos los esfuerzos que se requieren para posicionar cada uno de los pivotes que estén involucrados en la ordenación, en la ubicación que les corresponde. Por simplicidad, se considera que $m = 2^j - 1$, los particionamientos son perfectamente balanceados, $k = 2^h - 1$, $h \leq j$, los índices del arreglo van de $0 \ldots m-1$ y la Figura 4.

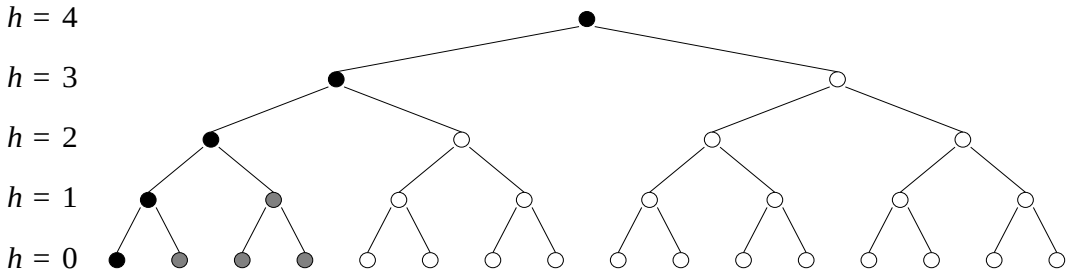


Figura 4: Árbol de particionamiento de un arreglo con balanceo perfecto.

Desde el punto de vista de la cantidad de niveles del árbol de particionamiento, posicionar correctamente un pivote de un nivel h requiere de $2^{h+1} - 2$ comparaciones (con las cuales particiona su subarreglo respectivo de $2^{h+1} - 1$ elementos).

Note que en el proceso de ordenamiento, de cada nivel se consideran sólo los primeros $\lceil \frac{k}{2^{h+1}} \rceil$ elementos. Por ejemplo, para ordenar hasta $k = 7$, encontrar al menor del conjunto requiere posicionar todos los elementos negros de la Figura 4, y luego, para la selección de los elementos $1 \ldots 6$, se requiere posicionar a los elementos grises. Note que en el caso de los elementos grises, se suma todo el trabajo de ordenar el conjunto de k menores, en cambio para los elementos negros, sólo se considera el esfuerzo del particionamiento.

El trabajo total se muestra en la Ecuación (1). La primera sumatoria corresponde a la ordenación de los primeros k elementos, y la segunda, al trabajo realizado al encontrar los pivotes que no fueron considerados al buscar al menor del conjunto completo.

$$T(k) = \sum_{h=0}^{\lfloor \log_2 k \rfloor} \lceil \frac{k}{2^{h+1}} \rceil \left(2^{h+1} - 2\right) + \sum_{h=\lfloor \log_2 k \rfloor + 1}^{\lfloor \log_2 m \rfloor} 2^{h+1} - 2 = O(k \log k) + O(m) \tag{1}$$

Por lo tanto, el esfuerzo total de la ordenación en línea hasta el k-ésimo elemento, es de la forma $C_1 \cdot m + C_2 \cdot k \log k$. En el caso que se ordene todo el conjunto se llega a una complejidad esperada de $O(m \log m)$. Se realizaron experimentos preliminares que con permiten confirmar nuestra conjetura. En un trabajo posterior se demostrará esta conjetura.

4. Optimización del Algoritmo de Kruskal utilizando Ordenación en Línea

Al igual que al optimizar con heaps, la idea es restringir la ordenación de aristas sólo a las necesarias. La optimización consiste en seleccionar incrementalmente las aristas de acuerdo a su peso utilizando el algoritmo IQS, evitando de este modo la ordenación inicial de las aristas. También se tiene que en el peor caso se necesitan revisar todas las aristas del grafo. El algoritmo de Kruskal optimizado con ordenación en línea se muestra en la Figura 5.

```
Kruskal3 (Grafo G(V, E))

  UnionFind C ← {{v}|v ∈ V} // el conjunto de las componentes conexas
  ACM ← ∅ // árbol cobertor mínimo
  Pila p
  p.push(|E|)
  int i ← 0
  while |C| > 1 do
    IQS(E, i, p)
    (e = {v, w}) ← E[i] // la siguiente arista en orden creciente de peso
    if C.find(v) ≠ C.find(w) then
      ACM ← ACM ∪ {e}
      C.union(v, w)
    i++
  return ACM
```

Figura 5: Algoritmo de Kruskal optimizado con ordenación en línea.

La inicialización de UnionFind toma tiempo $O(n)$. Se realizan a lo más m iteraciones del ciclo while, sea k la cantidad de iteraciones realizadas ($k \leq m$). En la primera iteración se selecciona a la arista de menor costo, lo que toma tiempo esperado $C_1 \cdot m$; en las siguientes iteraciones se

selecciona la arista de costo mínimo, que en conjunto tienen costo de la forma $C_2 \cdot k \log k$. Todas las operaciones de union y find en conjunto tienen costo $O(k)$. Luego, el tiempo esperado de este algoritmo es $C_1 \cdot m + C_2 \cdot k \log k$, y la complejidad de peor caso ($k = m$) es $O(m \log m)$.

5. Resultados Experimentales

Se realizaron pruebas sobre grafos sintéticos cuyas aristas tienen costos generados aleatoriamente según una distribución uniforme en el intervalo [0,1]. Se dejaron libres los siguientes parámetros: la cantidad de vértices y la densidad de aristas del grafo. Las aristas de los grafos fueron seleccionadas uniformemente. Para cada grafo se calculó el ACM utilizando los 3 algoritmos, restituyendo el grafo a su estado inicial antes de utilizar el siguiente algoritmo. En el caso de grafos completos, también se midieron los resultados del algoritmo de Prim.

Los rangos de variación de los paramétros son: $n \in [1{,}000 \ldots 10{,}000]$ y $densidad = [0.5\,\%, 100\,\%]$, donde $densidad = \frac{2m}{n(n-1)} 100\,\%$. Con la variación de la densidad se pretende estudiar el comportamiento de las optimizaciones a medida que el grafo posee más aristas.

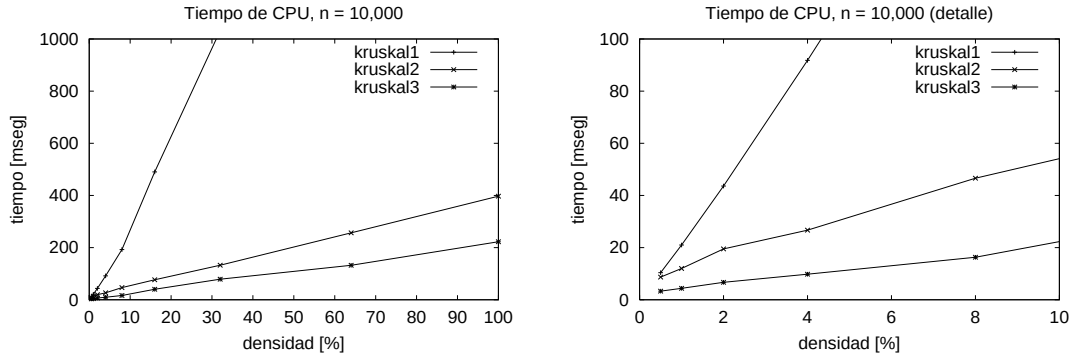
La máquina utilizada para las pruebas está equipada con un procesador Intel Pentium de 2.0 GHz, 512 MB RAM y disco local. Para medir el desempeño de los algoritmos interesa obtener el tiempo de CPU que estos utilizan. Adicionalmente se midió la cantidad de iteraciones realizadas durante el ciclo while de los algoritmos (que corresponden a la cantidad de aristas que se necesitan ordenar), no se muestra este resultado pues como los tres algoritmos utilizaban el mismo grafo realizan la misma cantidad de iteraciones, hecho que se verificó en la fase experimental de este trabajo. En las pruebas hechas con el algoritmo de Prim, sólo se midió tiempo de CPU.

Por economía llamaremos kruskal1 al algoritmo de Kruskal básico, kruskal2 al algoritmo de Kruskal optimizado con heaps, kruskal3 al algoritmo de Kruskal optimizado con ordenación en línea y prim al algoritmo de Prim.

Se intentó realizar pruebas con [3] para lo cual se obtuvieron los códigos empleados por sus autores (desde `http://www.mpi-sb.mpg.de/ sanders/dfg/iMax.tgz`). Lamentablemente, debido a que las estructuras empleadas por los autores de [3] requieren demasiada información adicional, no se pudo completar la serie de pruebas debido al excesivo uso de recursos (requiriendo 1.5 y 1.9 GB de RAM para 9,000 y 10,000 nodos respectivamente, al instante de detener el experimento). Adicionalmente, se verificó que los tiempos no eran competitivos ni contra prim, ni kurskal3.

La Figura 6 (a) muestra una comparación de las tres versiones para grafos de $n = 10{,}000$ vértices, variando la $densidad \in [0.5\,\%, 100\,\%]$. Se aprecia que kruskal1 es, por mucho, más costoso, y ambas optimizaciones mejoran el tiempo de procesamiento, siendo sistemáticamente mejor kruskal3. También se aprecia que a medida que el grafo se hace más denso es más notoria la ventaja de kruskal3 sobre los otros algoritmos. La Figura 6 (b) muestra un detalle de la comparación para $densidad \in [0.5\,\%, 10\,\%]$.

Las Figuras 7 (a), (b), (c) y (d) muestran la comparación para grafos de distintas densidades: 1 %, 4 %, 32 % y 100 % respectivamente, con lo que se quiere mostrar el comportamiento para grafos ralos, de densidad media y completos, variando la cantidad de vértices. Al considerar los cuatro gráficos se observa que kruskal3 es sistemáticamente mejor en todas las condiciones de cantidad de vértices y densidad de aristas. Como era de esperar, la mejora de tiempo para grafos ralos no es tan significativa como a densidades mayores, por ejemplo para $densidad = 1\,\%$, se observa que kruskal3 es 3.02 veces más rápido que kruskal1 y 1.74 veces más rápido que kruskal2. Esta mejora de rendimiento de kruskal3 aumenta considerablemente frente a kruskal1 a medida que la



(a) (b)

Figura 6: Tiempo de CPU variando la densidad de aristas del grafo, $n = 10{,}000$. (a) Kruskal1 llega a 3,500 mseg con densidad de 100 % (grafo completo). (b) Detalle del gráfico para $densidad \in [0.5\,\%, 10\,\%]$.

densidad del grafo es mayor, también se notan mejoras frente a kruskal2 cuando la densidad de arcos aumenta. Por ejemplo, para una $densidad = 4\,\%$ se aprecia que kruskal3 es 7.07 veces más rápido que kruskal1 y 2.29 veces más rápido que kruskal2; y para $densidad \geq 32\,\%$ se aprecia que kruskal3 es 13.13 veces más rápido que kruskal1 y 2.14 veces más rápido que kruskal2. En (d), donde se realizan pruebas con grafos completos, adicionalmente se muestran los tiempos que toma la construcción del árbol cobertor mínimo utilizando prim, se aprecia que para todo el rango de valores en la cantidad de nodos, kruskal3 se comporta mejor que prim.

En la Tabla 1 mostramos los modelos de regresión para las tres versiones del algoritmo de Kruskal. Los modelos se escogieron según el análisis realizado para cada algoritmo, especificándose en función de $m = |E|$ y k la cantidad de iteraciones realizadas, que se corresponde con el tamaño del subarreglo de aristas livianas a revisar. La tabla muestra como el ordenamiento en línea es un mejora fuerte para el algoritmo de Kruskal (recuerde que un grafo completo no dirigido de 5,000 vértices tiene 12.5 millones de aristas, y $log_2(12.5 \cdot 10^6) = 24$) y que también es una mejora frente a la optimización utilizando heaps.

	Kruskal básico	Kruskal optimizado Heaps	Kruskal optimizado Ordenamiento en Linea
Modelo	$3{,}45 \cdot m \log(m)$	$7{,}62 \cdot m + 18{,}90 \cdot k \log(m)$	$3{,}93 \cdot m + 6{,}01 \cdot k \log(k)$
Correlación	98.54 %	99.94 %	98.29 %

Cuadro 1: Complejidades empíricas de los algoritmos en función de $m = |E|$ y k el tamaño del subconjunto de aristas revisadas. El tiempo se mide en nanosegundos.

6. Conclusiones

En este trabajo se presenta una optimización del algoritmo de Kruskal basada en un algoritmo de ordenación en línea, competitiva frente a la versión tradicional del algoritmo de Kruskal y su optimización utilizando Heaps. La motivación de este trabajo es mostrar la técnica de ordenación en línea, que según los análisis y experimentos preliminares que hemos realizado, permite ordenar los primeros k elementos de un arreglo, con k desconocido, utilizando el mismo tiempo que toma

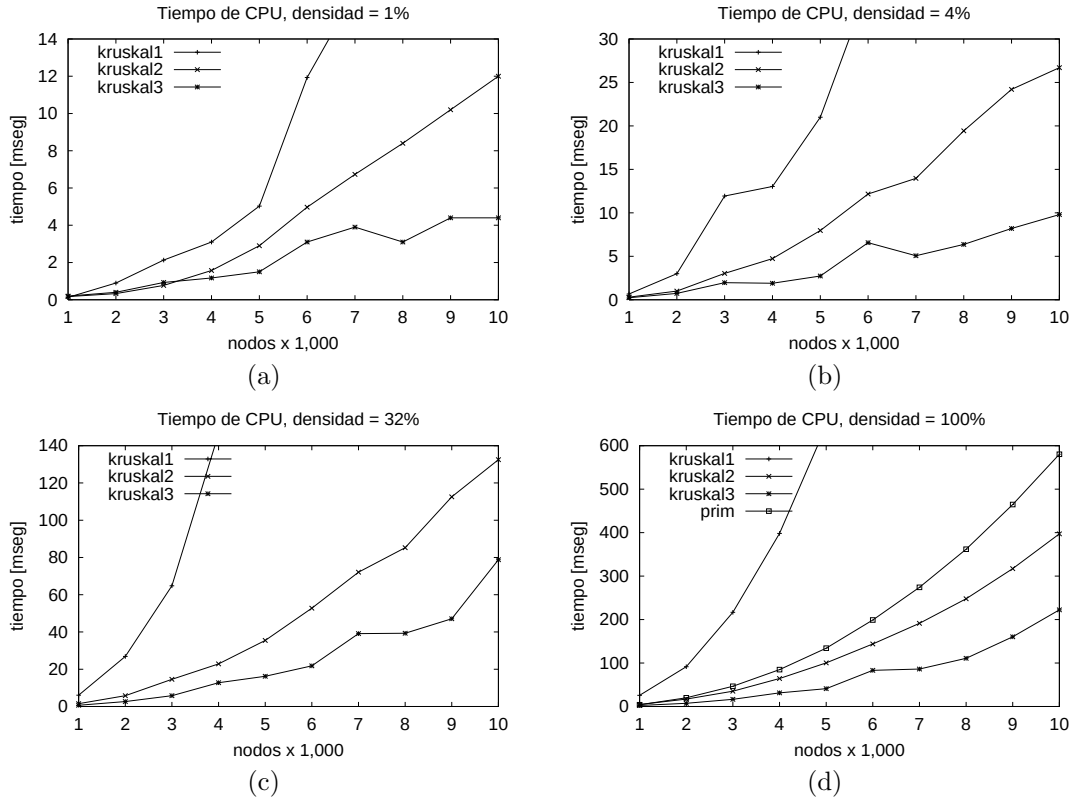


Figura 7: Tiempo de CPU variando la cantidad de nodos para distintas densidades. (a) Para 10,000 nodos y $densidad = 1\,\%$, kruskal1 llega a 21 mseg. (b) Para 10,000 nodos y $densidad = 4\,\%$, kruskal1 llega a 92 mseg. (c) Para 10,000 nodos y $densidad = 32\,\%$,kruskal1 llega a 1032 mseg. (d) Para 10,000 nodos y $densidad = 100\,\%$, kruskal1 llega a 3500 mseg.

el ordenar los k menores elementos de un arreglo cuando k es un valor conocido.

Utilizando la técnica de ordenamiento en línea se obtuvieron tiempos de CPU de la forma $C_1 \cdot m + C_2 \cdot k \log k$ para el algoritmo de Kruskal, donde en la práctica el término dominante es el que tiene que ver con el tamaño del conjunto de arcos, lo cual es una mejora significativa para este algoritmo. Por ejemplo, para grafos ralos esta versión es tres veces más rápida que el algoritmo básico, y para grafos de densidad media o grafos completos la óptimización llega a ser 13 veces más rápida que el algoritmo básico. Como se ve de los resultados experimentales, el algoritmo de Kruskal optimizado con ordenación en línea puede ser empleado tanto para grafos ralos como para grafos densos. En particular, para grafos completos esta optimización tiene mejor desempeño que el algoritmo de Prim.

Dentro de los próximos pasos de nuestra investigación está demostrar la complejidad de caso promedio del algoritmo de ordenación en línea.

Referencias

[1] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[3] I. Katriel, P. Sanders, and J. L. Träff. A practical minimum scanning tree algorithm using the cycle property. Research Report MPI-I-2002-1-003, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, October 2002.

[4] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[5] B. M. E. Moret and H. D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Proc. 2nd Workshop Algorithms and Data Structures (WADS'91)*, LNCS 519, pages 400–411, 1991.

[6] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[7] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983.

[8] M. A. Weiss. *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, 1995.