Evaluación de la Eficiencia del Algoritmo de Kruskal para la Construcción de Árboles Cobertores Mínimos

María T. Ortega, Mario A. Ramos, estudiantes
Universidad Tecnológica de Panamá
maria.ortega2@utp.ac.pa, mario.ramos@utp.ac.pa

Resumen- *Este artículo brinda un análisis de la eficiencia de cuatro variantes del algoritmo de Kruskal para la construcción de árboles cobertores mínimos de un grafo dado y el impacto de la utilización de algoritmos de búsqueda óptimos en el rendimiento esperado del algoritmo.*

Palabras Clave - Árbol cobertor mínimo, Algoritmos, Eficiencia, IQS, Kruskal, QuickSort.

1. Introducción

El desarrollo de software ha tenido un gran crecimiento en las últimas décadas, apoyándose en gran parte de los avances en las tecnologías de construcción de computadores. Esto ha permitido la creación de algoritmos y programas que resuelven problemáticas en diferentes dominios, apoyándose en la capacidad de procesamiento de los computadores y dejando de lado la eficiencia del propio algoritmo. Sin embargo, buenos algoritmos implementados, haciendo énfasis en la eficiencia, permiten sacar mayor partido al recurso computacional.

Una forma de evaluar la importancia de la implementación de algoritmos eficientes para la solución de una problemática en particular, es utilizar diferentes algoritmos que resuelvan la misma problemática y comparar sus características de tiempo de ejecución, consumo de recursos, complejidad de implementación, entre otras; lo cual puede servir de métrica para decidir la opción que mejor se adecúe a las necesidades requeridas.

Si se tienen algoritmos que resuelven la misma problemática de manera distinta, ¿cómo decidir cual algoritmo es mejor?, posibles respuestas a esta interrogante serían: el que conlleve menor tiempo de ejecución, menor complejidad de implementación y(o) menor consumo de recursos.

Uno de los problemas clásicos de las ciencias de la computación consiste en la búsqueda del árbol de cobertura de costo mínimo (MST, *Minimum Spanning Tree*) de un grafo conexo, no dirigido y con aristas con peso *G(V,E)*, en que V corresponde al conjunto de nodos y *E* al de aristas (o arcos). En este breve artículo se analizarán experimentalmente algunas soluciones para este problema.

2. Árbol Cobertor Mínimo

Sea un grafo conexo, no dirigido y con aristas con peso G(V,E), en que *V* corresponde al conjunto de nodos y *E* al de aristas.

El árbol de cobertura de costo mínimo (también llamado árbol de cobertura de mínimo costo) es un árbol que conecta a todos los vértices en *V* mediante las aristas contenidas en *E* con un costo total mínimo [1]. Siguiendo la notación estándar se considerará que los tamaños de los conjuntos de nodos y aristas son $n = |V|$ y $m = |E|$, respectivamente.

El MST es un árbol porque es conexo y acíclico, es de cobertura porque cubre todos los vértices y es de costo mínimo porque la suma de los costos de las aristas del MST es la menor posible.

En general, cualquier grafo no dirigido (sea conexo o no) tiene un bosque de cobertura de costo mínimo. Este bosque está compuesto por la unión de los distintos MSTs de cada una de las componentes del grafo procesadas independientemente.

El MST puede no ser único para un grafo G dado. Note que el número de aristas del MST es $|V| - 1$, por lo que un grafo con una alta densidad de aristas puede tener múltiples árboles cobertores mínimos de igual coste total.

Los algoritmos básicos para resolver este problema son de tipo *greedy* (algoritmos avaros), y los más conocidos son el algoritmo de Kruskal y el de Prim [1]. Usando el algoritmo de Prim, el cálculo del MST toma tiempo $O(n^2)$, lo cual lo hace más conveniente para grafos densos. A su vez, el algoritmo de Kruskal permite calcular el MST en tiempo $O(m \log m)$, que es apropiado para grafos dispersos. A continuación nos concentraremos en el algoritmo de Kruskal y algunas optimizaciones que se le pueden hacer de modo de mejorar su desempeño incluso en el caso de grafos densos.

3. Algoritmo de Kruskal

El algoritmo de Kruskal para construir árboles cobertores mínimos inicia su funcionamiento con n componentes que contienen un solo nodo, que son mezclados hasta producir una única componente conexa. Para la creación de esta única componente conexa, el algoritmo de Kruskal utiliza un par de estructuras auxiliares: la primera es C, que es una instancia del Tipo de Datos Abstractos (TDA) *UnionFind* [1], la que será explicada a continuación. La segunda estructura auxiliar es ACM que se inicializa en (V,φ), esto es un bosque de n árboles de un nodo, que representa al MST en construcción.

Kruskal opera iterativamente, y en cada iteración verifica si es posible insertar una nueva arista. El criterio de selección de arcos es escoger incrementalmente el arco de menor costo en E tal que no produce un ciclo con las aristas previamente almacenadas en ACM; esto significa, que sólo agrega arcos cuyos vértices pertenezcan a dos componentes distintas (es decir, dos árboles distintos del bosque). Una vez agregada la arista ambas componentes se fusionan en una sola (restando un árbol al bosque). Cuando el proceso termina, la estructura ACM contiene al árbol cobertor mínimo del grafo *G(V,E)* como se muestra en la Figura 1.

El TDA UnionFind permite administrar eficientemente una colección de conjuntos disjuntos permitiendo consultar a que conjunto pertenece un elemento dado (Find) o unir dos conjuntos distintos (Union). Durante la ejecución de Kruskal, se tiene exactamente este caso. Se inicia con conjuntos disjuntos y cada vez que se analiza una arista se determina si sus vértices pertenecen a componentes distintas, es decir, a conjuntos distintos para luego unir las componentes.

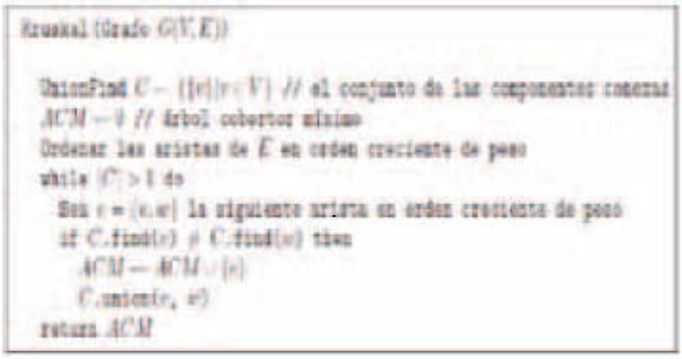

Figura 1. Algoritmo de Kruskal Básico. Adaptado de [1]

La versión estándar del TDA UnionFind se presenta en la Figura 2. Con esta estructura, la operación Find consiste en seguir los punteros (hacia el padre) hasta llegar a la raíz del árbol al que pertenece el nodo. En este caso la raíz es un nodo que se apunta a sí mismo. Una vez que se conoce la raíz, se puede actualizar el puntero de cada nodo en el camino hacia la raíz para dejarlo apuntando directamente a la raíz, lo que se conoce como compresión de caminos [1].

La operación Union fusiona las componentes. En donde el peor caso posible podría producir caminos de largo proporcional a la cantidad de nodos. Para prohibir la ocurrencia del peor caso, la idea es garantizar que la altura de los árboles sea logarítmica es decir, que contenga caminos relativamente cortos. La estrategia de fusión consiste en hacer que la componente que tenga menos elementos quede como hija del representante de la componente más numerosa. Tras fusionar se actualiza el tamaño de la componente resultante.

La implementación estándar de la estructura UnionFind considera dos valores *(r,c)* para cada identificador de nodo (Figura 2), en que *r* es el representante del nodo y *c* indica la cantidad de elementos en la componente.

4. Optimizaciones del Algoritmo de Kruskal

Para propósitos de este estudio se utilizarán cuatro variantes eficientes del algoritmo de Kruskal, a saber, Kruskal0, Kruskal1, Kruskal2 y Kruskal3. Kruskal0 corresponde a la implementación básica de Kruskal (Figura 1) en que las aristas se ordenan con Quicksort (Figura 4), se utiliza la versión de UnionFind mostrada en la Figura 2, y se deja explícita la condición de término del ciclo While con una operación que es más fácil de codificar. El seudocódigo de la versión básica se muestra en la Figura 3.

La segunda variante del Kruskal, denominada Kruskal1, incorpora el algoritmo de ordenación incremental IQS (Incremental QuickSort) (Figura 5) [2]. IQS permite obtener los primeros *k* elementos en orden creciente de un conjunto desordenado A de tamaño m para cualquier valor de $k \leq m$, en tiempo óptimo O(m + k logk).

El algoritmo Kruskal1 al igual que Kruskal0 hace uso de la estructura estándar UnionFind (Figura 2). La variante del algoritmo de Kruskal1 con las adecuaciones necesarias para utilizar el algoritmo IQS se muestra en la Figura 6.

La tercera y cuarta variante del algoritmo de Kruskal,denominadas Kruskal2 y Kruskal3, utilizan los algoritmos de búsqueda Quicksort

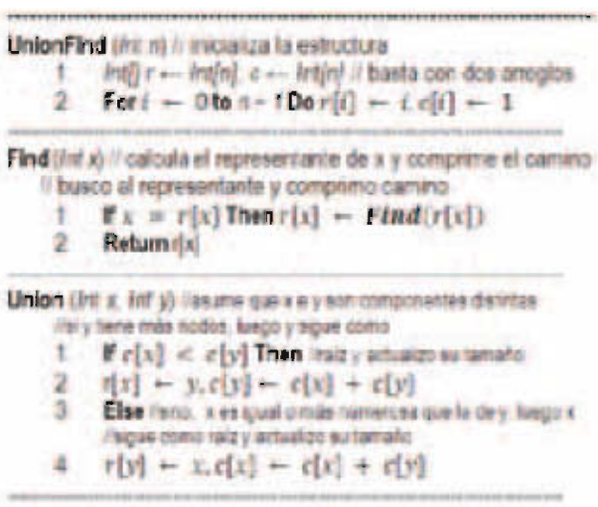

Figura 2. Estructura UnionFind estándar. Adaptado de [1]

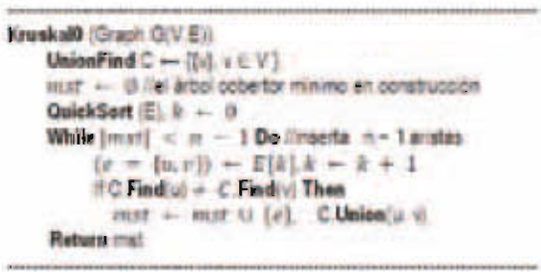

Figura 3. Algoritmo Kruskal0

```
QuickSort (A, p, r)
  if p ← r  then
q=Particion(A,p,r)
QuickSort(A,p,q)
QuickSort(A,q+1,r)
```

Figura 4. Algoritmo QuickSort [3]

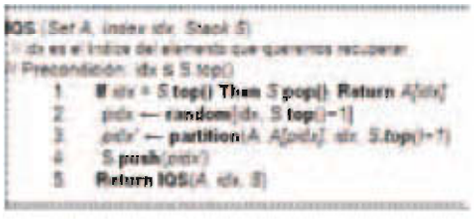

Figura 5. Algoritmo Incremental IQS [2]

(Figura 4) e IQS (Figura 5) respectivamente, pero utilizan una implementación alternativa de la estructura UnionFind (Figura 7). Esta variante utiliza una variante no recursiva de la función Find, para ello utiliza una Pila que permita actualizar los índices de los representantes de cada nodo.

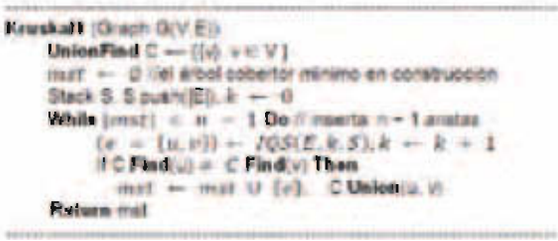

Figura 6. Algoritmo Kruskal1

```
FindInd (int x)
        While(x!=r[x]
  S.Agregar(x)
x ← r[x]  then
      While(S.Top()!=null)
            r[S.Sacar]=x
retorna x;
```

Figura 7. Estructura alternativa de Find. Adaptado de [1]

5. Metodología Experimental

En este artículo se evalúa el impacto de implementaciones eficientes de las estructuras UnionFind en conjunto con los algoritmos de búsqueda en los tiempos de construcción del árbol cobertor mínimo utilizando el algoritmo de Kruskal.

Para el estudio experimental se utilizarán grafos Euclidianos conexos no dirigidos representados bajo una lista de arcos.

Los grafos Euclidianos son grafos ponderados en los cuales, los vértices están conformados por puntos pertenecientes al plano Euclidiano R2 [4], y cuyas aristas son ponderadas en relación a las distancias Euclidianas entre dos vértices del grafo.

Para propósitos de este estudio se considera un grafo euclidiano no dirigido G(V,E), como un conjunto de puntos (x,y) dentro del cuadrado unitario [0,1] × [0,1] donde el costo de los arcos del grafo equivale a la distancia euclidiana d (1) entre dos vértices v, denotados por vk(xk,yk), del grafo dado. Los arcos considerados en el grafo son elegidos aleatoriamente, iniciando con grafos con baja densidad de arcos hasta obtener el grafo completo.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \qquad (1)$$

El tamaño del conjunto de arcos E (2) pertenecientes a un grafo G, estará delimitado por la densidad de arcos p, en donde un grafo con una densidad p = 0.01 contendrá el 1% del total de arcos posibles, mientras que un grafo con densidad p = 1 contendrá el 100% del total de arcos posibles es decir, el grafo completo.

$$|E| = p\frac{n(n-1)}{2} \qquad (2)$$

El análisis realizado en este artículo se basa en un estudio comparativo de las cuatro variantes del algoritmo Kruskal, explorando distintos valores de densidad de arcos y número de nodos. Se midió el tiempo de construcción del MST para cada una de las versiones de Kruskal, identificando la más eficiente en tiempo de ejecución.

6. Resultados y Conclusiones

Las diversas variantes del algoritmo de Kruskal, junto con la representación del grafo Euclidiano, fueron implementadas en el lenguaje Java, y las pruebas experimentales se realizaron en una máquina con las siguientes especificaciones: procesador Intel Pentium M 740 2.66GHZ, memoria 2 GB DDR2 RAM, sistema operativo Windows 7 Ultimate, Java SE Development Kit 6 Update 17, NetBeans IDE 6.7.1 y se configuró la máquina virtual de java con un heap size de = -Xms500m -Xmx1536m.

Se realizaron pruebas con densidad de arcos p correspondientes a 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.0, y una cantidad de nodos variable de 1,000 a 10,000 nodos. Las pruebas ejecutadas con densidades de 0.01 a 0.16 fueron ejecutadas hasta un máximo de 10,000 nodos; para la densidad de 0.32 las pruebas alcanzaron un tope con 7,000 nodos, con la densidad 0.64 se alcanzó ejecutar hasta un máximo de 5,000 nodos y con la densidad de 1.0 alcanzó el límite con 4,000 nodos. Estos límites corresponden al máximo nivel de memoria y procesamiento soportados utilizando un computador con las especificaciones anteriormente mencionadas.

La Figura 8 muestra los resultados obtenidos de la prueba de las cuatro variantes de Kruskal, con una densidad de 0.01 y valores de n variables de 1,000 a 10,000 nodos con un incremento de 1,000. La gráfica muestra una representación por tiempo vs nodos, ponderados en valores de milisegundos y múltiplos de 1,000 respectivamente. Las implementaciones Kruskal 1 y Kruskal 3 son notablemente más rápidas que las implementaciones de Kruskal 0 y Kruskal 2. Esto se debe a la influencia directa del algoritmo de ordenamiento utilizado, puesto que Kruskal 1 y 3, utilizan el algoritmo IQS, el cual es más eficiente que el algoritmo Quicksort para ordenar las aristas que son necesarias para construir el MST.

Otro punto importante es comparar las diferencias entre las implementaciones de la estructura UnionFind. Para ello se muestran los resultados obtenidos de la comparación de los algoritmos Kruskal1 y Kruskal3 con densidad de 0.32 (Figura 9). Puesto que ambos utilizan el algoritmo de búsqueda IQS, la pequeña diferencia de tiempos se debe a la implementación de la estructura UnionFind. Puesto que el algoritmo Kruskal3 utiliza la implementación alternativa FindInd, puede concluirse que esta variante es más eficiente que la estructura estándar.

Las implementaciones de Kruskal 1 y 3, resultaron ser las más eficientes debido a la utilización del algoritmo incremental IQS. En el peor de los casos, cuando se necesita ordenar el conjunto completo, el costo derivado de ejecutar IQS resulta ser de O(m logm). Sin embargo, debido a que en la práctica Kruskal sólo utiliza una cantidad pequeña de aristas para conectar a los nodos (en efecto, de acuerdo a [5], sólo requiere revisar las primeras k = O(n ln(n)) aristas) en promedio IQS consume O(m + log2n) [2], que en la práctica es bastante menor que O(m logm).

Figura 8. Resultados de prueba con densidad de 0.01

Las tendencias observadas en las pruebas con cada una de las densidades de arcos y con cada bloque de nodos utilizado denotan que los tiempos menores de creación del MST corresponden a las implementaciones que utilizan el IQS (Fig. 9) como algoritmo de ordenamiento, siendo el ordenamiento la operación de mayor costo del algoritmo y un punto importante de optimización. Esto es consistente con los resultados obtenidos en [6], en donde también IQS muestra el mejor resultado para este tipo de aplicaciones.

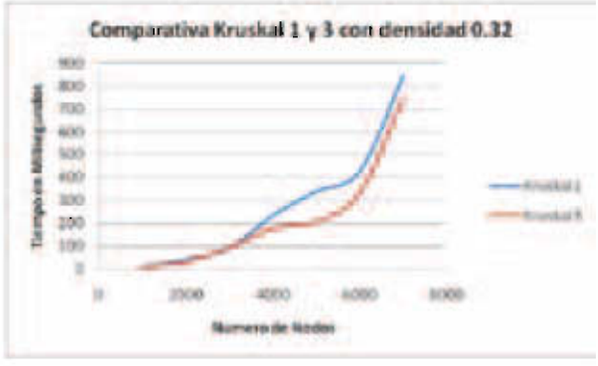

Figura 9. Resultados de prueba con densidad 0.32

Las implementaciones de las estructuras UnionFind, influyen en menor grado en la eficiencia de los algoritmos siendo ligeramente más eficientes los que utilizan la estructura alternativa no recursiva (FindInd) a la compresión de caminos estándar (UnionFind) siempre y cuando utilicen el mismo algoritmo de ordenamiento. Esta tendencia pudo observarse a lo largo de las pruebas para todas las densidades de arcos y cantidades de nodos (vértices), tal como se muestra en la Figura 10.

Figura 10. Resultados de pruebas con densidad 1.0

Agradecimientos

Deseamos agradecer al Profesor Rodrigo Paredes por sus comentarios sobre este trabajo y su colaboración en la redacción del presente artículo.

Referencias

[1] T. H. Cormen, C. E. Leiserson and R. L Rivest, and C. Stein. "Introduction to Algorithms". 2ª ed. McGraw Hill 2001.
[2] R. Paredes. "Graphs for Metric Space Searching". Universidad de Chile, Santiago, Chile, Julio 2008.
[3] R. Guerequeta and A. Vallecillo, Técnicas de diseño de algoritmos, 2a ed., Publicaciones de la Universidad de Málaga, Mayo 2000.
[4] J.B. Hayet, "Caminos más cortos", Centro de Investigación en Matemáticas, Guanajuato, México, 2008.
[5] S. Janson, D.Knuth, T. Łucsak and B. Pittel. "The birth of the giant component." Random Structures & Algorithms, 4(3):233-358,1993.
[6] R. Paredes, "Aplicación de Ordenamiento en Línea: Construcción Eficiente del Árbol Cobertor Mínimo", in Proc. Conference on Informatics (CLEI'03), 2003.