**Análisis y diseño de algoritmos II**

**Trabajo Practico Especial**

**Algoritmo Prim con Heap**

**Integrante 1:** Leandro Salias

* **E-Mail:** leandrosalias42@gmail.com

**Integrante 2:** Gonzalo Clementi

* **E-Mail:** gonzacl26@gmail.com

Índice

**Introducción………………………………………………………………………………….…………3**

* Introducción del problema que se implementa…………………………….…**3**

**Decisiones de diseño de la implementación……………………………………….……3**

* Diseño e implementación…………………………………………………………………**3**

**Clases…………………………………………………………………………………………….…………3**

**Análisis de la complejidad…………………………………………………………………………4**

* Análisis de la complejidad de los algoritmos implementados……………**4**

**Conclusión……………………………………………………………………………………..…………6**

* Análisis del comportamiento de los algoritmos

según los casos de entrada……………..….………………………………….….…….**6**

* Ejemplo 1……………………………………………………………………………………………………**6**
* Ejemplo 2……………………………………………………………………………………………………**7**
* Mejoras propuestas……………………………………..……………………………….….**8**

**Referencias………………………………………………………………………………………….……8**

Introducción

Para este trabajo se le propuso a nuestro grupo implementar el algoritmo de Prim utilizando como estructura auxiliar un Heap (o cola de prioridad). Este algoritmo perteneciente a la teoría de grafos permite encontrar un árbol de recubrimiento mínimo de un grafo conexo, no dirigido y con peso o costo en sus aristas. En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

El algoritmo fue diseñado en 1930 por el matemático Vojtech Jarnik y luego de manera independiente por el científico computacional Robert C. Prim en 1957 y redescubierto por Dijkstra en 1959.

En este informe se presentará la idea de diseño del algoritmo y sus estructuras auxiliares, se analizará la complejidad del algoritmo en base a lo implementado y se propondrán posibles mejoras al costo final del mismo.

Diseño e implementación

Para implementar la clase Grafo utilizamos como base la propuesta por la cátedra y como estructura principal decidimos que esté representado con un mapa de vértices que a su vez cada uno contiene un mapa de arcos, los cuales almacenan el vértice adyacente y su costo.

Por otro lado se nos pidió implementar el algoritmo de Prim utilizando un Heap como estructura auxiliar, por lo que decidimos implementar una clase Heap que funcione como un MinHeap (el elemento con mayor prioridad es el de valor mínimo), para que almacene los vértices y sus respectivas distancias. Como lo que debe almacenar son pares (vértice, distancia) decidimos crear una clase Par dentro de la clase Heap (algo parecido a la clase Arco dentro del Grafo).

Como estructura principal utilizamos un vector (STL) que almacena los elementos de manera tal que éste funcione como un árbol binario. Luego para reducir el costo de algunas funciones de búsquedas decidimos agregar a la estructura un mapa que guarde para cada elemento su posición en el arreglo (por ejemplo, en la función Existe para reducir su costo de O(n) a O(log n), siendo n la cantidad de elementos).

Clases

La clase Par funciona de manera tal que permite almacenar en una variable de este tipo, un elemento parametrizado (en nuestro caso van a ser los vértices) y su costo o distancia. Para diseñarla nos basamos en la clase Arco incluida en la clase Grafo ya que funcionan de manera similar. Sus métodos tienen complejidad O(1) ya que devuelven un dato o realizan una asignación, y éstos son: devolver\_vertice, devolver\_dist (devuelve el valor de la distancia) y modificar\_dist (modifica el valor de la distancia).

La clase Heap guarda los elementos que se van agregando a la estructura de manera tal que el de mayor prioridad quede en el tope. Las funcionalidades a destacar son:

* Insertar: inserta un elemento al final del arreglo y luego se lo “flota” hasta que quede ordenado. Este método tiene complejidad O(log n), n = cant. Elementos.
* ExtraerMin: devuelve el valor de mayor prioridad en el Heap, en este caso el de mínimo valor, y lo elimina de la estructura. Luego coloca el último elemento en su posición y lo “hunde” hasta que el Heap quede ordenado. Su complejidad es O(log n), n = cant. Elementos.
* Actualizar: modifica el elemento en su campo de distancia o costo, y luego “hunde” o “flota” según corresponda, hasta que el Heap quede ordenado. La complejidad de este método es O(log n), n = cant. Elementos.
* Hundir: dada la posición de un elemento, lo reordena en la estructura de manera tal que le baja la prioridad hasta donde sea necesario. Complejidad temporal: O(log n), n = cant. Elementos.
* Flotar: dada la posición de un elemento, y la posición de su padre, lo reordena en la estructura de manera tal que le sube su prioridad hasta donde sea necesario. Complejidad temporal: O(log n), n = cant. Elementos.
* Existe: de tipo booleano, devuelve si un elemento dado existe o no en la estructura. Para realizar la búsqueda se utiliza la estructura auxiliar del mapa que guarda las posiciones de los elementos del arreglo, bajando la complejidad de la búsqueda de O(n) a O(log n).

Luego implementamos métodos internos como get\_hijoizq, get\_hijoder, get\_padre, éstos para obtener los distintos “nodos” del Heap, y el Intercambiar que intercambia dos elementos dadas sus posiciones. También algunos métodos públicos como esVacia, que retorna si el Heap está vacío; get\_tamanio, que devuelve la cantidad de elementos del Heap; y por último el Tope, que retorna el Par de mayor prioridad. Todos éstos últimos métodos tienen complejidad O(1) ya que sólo retornan o realizan asignaciones.

Análisis de la complejidad

En su versión original (implementación con vectores) el algoritmo de Prim posee una complejidad temporal de como mínimo O(V2) siendo V la cantidad de vértices del grafo.

En nuestro caso, al implementar la clase Heap sus métodos principales poseen como máximo una complejidad de O(log n), siendo n la cantidad de elementos agregados al Heap. Esto influye a la hora de calcular la arista de menor peso, ya que para la implementación con arreglos la operación para buscar ésta tiene complejidad lineal, y en este caso específico que se nos presentó logramos obtener una complejidad logarítmica.

Por otra parte, cabe aclarar que la implementación con Heap como estructura auxiliar sirve más que nada para grafos ralos, ya que si fueran más densos sería mucho más costoso al poseer más arcos y vértices, y en ese caso sería mejor utilizar la implementación con arreglos.

Analizando el pseudocódigo:

*Prim* (Grafo *G*)

*/\* Inicializamos todos los nodos del grafo.*

La distancia la ponemos a infinito y el padre de cada nodo a NULL

Encolamos, en una cola de prioridad

donde la prioridad es la distancia,

todas las parejas <nodo, distancia> del grafo\*/

**por cada** *u* en *V[G]* **hacer**

distancia[*u*] = INFINITO

padre[*u*] = NULL

Añadir(cola,<*u*, distancia[*u*]>) O(log n)

distancia[*u*]=0

**mientras** !esta\_vacia(cola) **hacer**

*// OJO: Se entiende por mayor prioridad aquel nodo cuya distancia[u] es menor.*

*u* = extraer\_minimo(cola) //devuelve el mínimo y lo elimina de la cola. O(log n)

**por cada** **v** adyacente a '**u'** **hacer** O(|E|)

**si** ((*v* ∈ cola) **&&** (distancia[*v*] > peso(u, v)) **entonces**

padre[*v*] = *u*

distancia[*v*] = peso(u, v)

Actualizar(cola,<*v*, distancia[*v*]>) O(log n)

Podemos concluir que la complejidad temporal del algoritmo es de O(|E|log(V)), siendo |E| la cantidad de aristas y V la cantidad de vértices.

En nuestro caso decidimos construir un grafo para el árbol de recubrimiento mínimo. A medida que se extraen los mínimos del Heap vamos agregando los vértices al grafo, lo cual tiene complejidad O(log n), por lo tanto no afecta la complejidad final del algoritmo. Finalmente devolvemos un grafo que será nuestro árbol de recubrimiento mínimo para el grafo original.

Luego para calcular el costo del árbol de recubrimiento mínimo decidimos implementar el DFS para grafos conexos, el cual recorre todas las aristas desde el primer vértice del árbol hasta el último sumando el costo de cada arista para devolverlo al finalizar el recorrido. Éste tiene complejidad temporal O(|V|+|E|) siendo |V| la cantidad de vértices y siendo |E| la cantidad de aristas.

Conclusión

Análisis del comportamiento de los algoritmos

A continuación, mostramos dos seguimientos de dos ejemplos de grafos con el resultado de su respectivo árbol de recubrimiento mínimo que devuelve el algoritmo Prim.

1

4

1

10000

3

**Ejemplo 1:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteración | A | S | V – S | Arco Min Peso |
| 1 | { } | { } | {1,2,3,4} | - |
| 2 | { } | {1} | {2,3,4} | (u,v) = (1,2) |
| 3 | {(1,2)} | {1,2} | {3,4} | (u,v) = (2,3) |
| 4 | {(1,2), (2,3)} | {1,2,3} | {4,5} | (u,v) = (3,4) |
| 5 | {(1,2), (2,3), (3,4)} | {1,2,3,4} | {Ø} | - |

**Árbol de recubrimiento de mínimo costo:**

1

1

10000

**Costo total del Árbol de recubrimiento de mínimo: 3**

**Ejemplo 2:**

40

30

20

55

45

10

50

15

25

35

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteración | A | S | V – S | Arco Min Peso |
| 1 | { } | { } | {1,2,3,4,5,6} | - |
| 2 | { } | {1} | {2,3,4,5,6} | (u,v) = (1,2) |
| 3 | {(1,2)} | {1,2} | {3,4,5,6} | (u,v) = (2,6) |
| 4 | {(1,2), (2,6)} | {1,2,6} | {3,4,5} | (u,v) = (6,3) |
| 5 | {(1,2), (2,6), (6,3)} | {1,2,6,3} | {4,5} | (u,v) = (4,6) |
| 6 | {(1,2), (2,6), (6,3), (4,6)} | {1,2,6,3,4} | {5} | (u,v) = (3,5) |
| 7 | {(1,2), (2,6), (6,3), (4,6), (3,5)} | { 1,2,6,3,4} | {Ø} | - |

**Árbol de recubrimiento de mínimo costo:**

20

10

15

25

35

**Costo total del Árbol de recubrimiento de mínimo: 105**

Mejoras propuestas

Como una posible mejora proponemos implementar el Heap de Fibonacci en lugar del MinHeap ya que éste ayuda a mejorar la complejidad temporal del algoritmo de Prim, reduciendo la misma de O(|E| log(V)) a O(|E|+ V log (V)). En esta tabla podemos observar cómo mejora la complejidad temporal de los métodos del MinHeap al Heap de Fibonacci:

|  |  |  |
| --- | --- | --- |
|  | **Min-Heap** | **Montículo de Fibonacci** |
| Insertar | O(log n) | O(1) |
| Acceso Mínimo | O(1) | O(1) |
| Borrar mínimo | O(log n) | O(log n)\* |
| Disminuir Clave | O(log n) | O(1)\* |
| Borrar | O(log n) | O(log n)\* |
| Unión | O(m log(n+m)) | O(1) |

Por último, una buena propuesta sería comparar ésta implementación del algoritmo de Prim con la que utiliza como estructura auxiliar un vector para ver como mejora el funcionamiento del algoritmo con nuestra implementación en grafos poco densos, y también para ver cómo afecta negativamente al tiempo de resolución de nuestro algoritmo en grafos muy densos.

Referencias

* <https://es.wikipedia.org/wiki/Algoritmo_de_Prim>
* https://es.wikipedia.org/wiki/Mont%C3%ADculo\_de\_Fibonacci