UNAM

Facultad de Ciencias

Análisis de Algoritmos

2022-2

Practica 04

Arroyo Lozano Santiago

June 14, 2022

0.1 Proyecto Γ

1 Introducción

La practica se desarrolló en `python 3` con una implementación de el algoritmo de Prim para búsqueda de arboles de peso mínimo usando heaps binarios (minHeap).

La Estructura de Datos seleccionada fueron los Heaps Binarios, implementamos su versión con arreglos tal que la raíz esté en el índice 0, el elemento con índice i siempre tendrá como hijo el índice $2i+1$ y como derecho al $2i+2$ y finalmente su padre será $\lfloor \frac{i-1}{2} \rfloor$

Análisis de Complejidad Heap Binario
Las complejidades de nuestro minHeap están dados por sus algoritmos *heapify-up* y *heapify-down*, los cuales en la implementación llamamos coloquialmente *arriba* y *abajo*.

Ambos algoritmos tienen una complejidad $O(\log n)$ ya que en el peor de los casos se recorre toda la altura del Heap (hacia arriba o hacia abajo) que al ser binario competo es por definición $\lfloor \log_2 n \rfloor$.

Finalmente como cada elemento será un vértice con un índice asociado podremos acceder a ellos en tiempo constante. Lo que en resumen lleva que insertar, reordenar y eliminar elementos del heap sea a lo más $O(\log n)$.

2 Algoritmo

Basándonos en la siguiente versión de Prim utilizando Heaps Binarios encontrada en las notas:

Algoritmo PRIM

Entrada: Una gráfica $G=(V, A)$ conexa no trivial, con pesos en las aristas. Se tiene que $|V|= n$, $|A|= m$.
Un vértice inicial a
Objetivo: Determinar un árbol de expansión para $G=(V, A)$
Variables Locales:

```
Q: Priority_Queue;     // Cola de prioridades
v, w : Vértices;       // vértices auxiliares
e : arista;            // arista auxiliar
BEGIN
   v ← G.FirstVertex;        // Toma el primer vértice de G
   WHILE (v ≠ Nil-Vertex) Do  // Da valores iniciales a los atributos de los vértices.
     v.visit ← False;    v.pa ← NIL;    v.d ← MaxInteger;
     v.inT ← False;      v.inQ ← NIL;
     v ← G.NextVertex;       // Toma el siguiente vértice de G
   ENDWHILE

   a.visit ← True;    v.d ← 0;   // Da valores iniciales al vértice a
   Q.Create;                     // Crea la Cola de prioridades
   Q.Insert (a);                 // Mete al vértice a con a.d=0 a Q


  // Construye el árbol generador de peso mínimo
   WHILE ( Not Q.Empty)  Do
      v ← Q.DeleteMin;      // Toma el vértice de peso mínimo
      v.inT ← True;         // v es agregado al árbol T
      Lv ← G[v].ady         // lista de vértices adyacentes a v (vecinos)
      e ← Lv.FirstEdge;     // Toma el primer vecino
      WHILE ( e ≠ Nil-Edge ) Do  // Revisa los vecinos.
         w ← e.v_final;
         IF ( Not w.visit ) THEN
            w.visit ← True;   w.d ← e.costo;   w.pa ← v;
            w.inT ← False;
            Q.Insert (w);     w.inQ ← ^w  // Apuntar a w en la cola Q
          ELSEIF ( w.inT ) THEN  // Nada por hacer
            ELSEIF ( e.costo < w.d ) THEN  // Se encontró un mejor camino
               w.pa ← v;   Q.DKey (w, e.costo); // Decrementa la llave de w
         ENDIF //
         e ← Lv.NextEdge;  // Pasa al siguiente vecino.
      ENDWHILE //  ( e ≠ Nil-Edge )
   ENDWHILE //  ( Not Q.Empty)
END  // Fin Prim
```

Página 6 de 13

Asumiremos que dicho algoritmo nos regresa un conjunto T que contiene los aristas y sus pesos, algo como se muestra en las mismas notas de la siguiente forma:

$$T = \{ (v_5, v_3, \$1), (v_5, v_7, \$2), (v_8, v_7, \$2), (v_1, v_5, \$2), (v_1, v_0, \$1), (v_3, v_9, \$1), (v_3, v_2, \$2), (v_6, v_2, \$1), (v_1, v_4, \$2) \}.$$

Lo que haremos es aplicar Prim a cada componente conexa de la gráfica. Para no repetir vértices, lo que haremos es agregarles una marca v.prim al vértice que indica si ya pasó el algoritmo Prim sobre el vértice o no.

Pseudocódigo

Algorithm 1 Γ

```
1: procedure Γ(G(V, E) : Grafica)
2:     L_F ← [ ]
3:     for each v ∈ V(G)
4:         if not v.prim
5:             T ← ∅
6:             T ← Prim(v)
7:             L_F.append(T)
```

Análisis de Complejidad Prim
El primer ciclo **WHILE** toma tiempo $O(n)$ ya que cada operación requiere tiempo constante. El segundo ciclo **WHILE** depende del tamaño de Q y en el peor de los casos Q tendrá a todos los vértices a la vez; es decir, el ciclo iterará a lo más n veces.

Como estamos usando Heaps Binarios para representar la cola de prioridades, como mencionamos anteriormente las operaciones insertar, eliminar y reordenar costarán $O(\log n)$, entonces como requerimos

- n operaciones insertar: $O(n \log n)$
- n operaciones eliminar: $O(n \log n)$
- m operaciones reordenar: $O(m \log n)$

Así que la suma de todo es $O(m \log n)$

Análisis de Complejidad
En nuestro algoritmo Γ en esencia lo que haremos es ejecutar Prim k veces donde k es el número de componentes conexas de la gráfica. Y como sabemos que Prim nos toma $O(m \log n)$ esto da k veces $O(m \log n)$ que sigue siendo $O(m \log n)$.

3 Como ejecutar

En la raíz de la carpeta se encuentra un archivo llamado `entrada.txt`. Este archivo debe contener la entrada de la gráfica en un formato particular; la primera fila enlistará todos los vértices de la gráfica y en las filas subsecuentes contendrá una tercia de elementos separada por comas donde los primeros dos serán vértices de un arista y el tercer elemento será el peso del arista.

El archivo que se debe ejecutar es **practica4.py**. Sin embargo los archivos `clases.py` y `prim.py` son necesarios para el correcto funcionamiento del programa ya que en ellas se definen ciertas dependencias que usaremos para resolver el problema.

```
1,2,3,4,5,6,7,8,9
1,2,1
1,7,2
2,7,3
2,4,1
7,4,2
3,5,3
3,6,1
3,8,1
5,6,4
5,9,2
6,9,2
8,9,5
```

Figure 1: Ejemplo de entrada

4 Salida

La salida será una lista de conjuntos, donde cada conjunto contendrá los aristas que conformen el árbol generador de peso mínimo de cada componente conexa de la gráfica.

Además, en `practica4.py` lo que hacemos es imprimir los valores de dicha lista y conjuntos de la siguiente forma para demostrar el correcto funcionamiento del programa:

```
[Running] python -u "c:\Users\santi\Documents\practica4\practica4.py"
El bosque genarador de peso minimo es:
 [{('1', '7', '2'), ('1', '2', '1'), ('2', '4', '1')}, {('3', '6', '3'), ('9', '5', '2'), ('3', '8', '1'),
 ('6', '9', '2')}]
COMPONENTE CONEXA 1
Arista:  1 7 de peso 2
Arista:  1 2 de peso 1
Arista:  2 4 de peso 1
COMPONENTE CONEXA 2
Arista:  3 6 de peso 3
Arista:  9 5 de peso 2
Arista:  3 8 de peso 1
Arista:  6 9 de peso 2
```

Figure 2: Ejemplo de salida