Universidad de Talca
Facultad de Ingeniería
Departamento de Ciencias de la Computación

Informe Tarea 5

Grafos

Fecha: January 8, 2019
Ignacio Martínez Hernández
e-mail: imartinez17@alumnos.utalca.cl

1 Introducción

En este informe, implementaré los algoritmos de Kruskal para la construcción del árbol cobertor mínimo y de Dijkstra para el cómputo de caminos más cortos, con el fin de analizar su comportamiento.

2 Análisis del problema

Para poder implementar los algoritmos del árbol cobertor mínimo y cómputo de caminos más cortos, primero es importante tener un grafo euclidiano conexo. Por eso, lo primero que se hizo fue la implementación de un algoritmo simple en C++ que creará un grafo totalmente aleatorio dado los parámetros ingresados en un espacio euclidiano dentro de un cuadrado unitario $[0,1] \times [0,1]$ y luego convertirla en su representación de Lista de Adyacencia para que Kruskal o Dijkstra lo utilicen. Después se pasó a implementar los algoritmos utilizando como guía Grafos para búsqueda en espacios métricos.

3 Solución del problema

3.1 Creación del grafo

3.1.1 Algoritmo de solución

- Se calculó el número de conexiones que se utilizarán con la fórmula $densidad \cdot (\frac{n \cdot (n-1)}{2})$.
- Se calcula $\Delta = \frac{1}{n}$ el cual se usa para generar un arreglo con distancias uniformes en las coordenadas x, y.
- Se hace un shuffle a los arreglos y luego se emparejan creando un nuevo arreglo con puntos aleatorios en $[0,1] \times [0,1]$.
- Se crea una matriz de adyacencia la cual va a contener -1 (indicando que no existe conexión) y 0 en sus diagonales (sólo para indicar que la distancia del nodo a si mismo es 0).

- En la matriz de adyacencia se le asignan conexiones para que sea conexa utilizando el arreglo que contiene las coordenadas.
- Después de tener una matriz de adyacencia conexa se generan más conexiones para asegurar la densidad.
- Por último, la matriz de adyacencia se convierte a una lista de adyacencia para luego ser utilizada por los respectivos algoritmos.

3.1.2 Implementación

input: n *densidad*
$edgesN \leftarrow densidad \cdot (\frac{n \cdot (n-1)}{2})$
$nodes, x, y, w \leftarrow \emptyset$
$\Delta \leftarrow \frac{1}{n}$
for $i \leftarrow 0$ ***to*** n **do**
  $x \leftarrow x \cup \Delta \cdot (i+1)$
  $y \leftarrow y \cup \Delta \cdot (i+1)$
end
suffle(x) , **suffle**(y)
for $i \leftarrow 0$ ***to*** n **do**
  $nodes \leftarrow nodes \cup (x_i, y_i)$
end
for $i \leftarrow 0$ ***to*** n **do**
  for $j \leftarrow 0$ ***to*** n **do**
    if $i == j$ **then**
      $w_{i,j} \leftarrow 0$
    else
      $w_{i,j} \leftarrow 1$
    end
  end
end
for $i \leftarrow 0$ ***to*** $n-1$ **do**
  $w_{i,i+1} \leftarrow w_{i+1,i} \leftarrow$**distance**$(nodes_i, nodes_{i+1})$
end
while $i < edgesN$ **do**
  $x' \leftarrow$ **random()**
  $y' \leftarrow$ **random()**
  if $x' \neq y' \wedge w_{x',y'} < 0$ **then**
    $w_{x',y'} \leftarrow w_{y',x'} \leftarrow$**distance**$(nodes_{x'}, nodes_{y'})$
    $w_{x',y'} \leftarrow w_{y',x'} \leftarrow 0$
  end
end

3.2 Kruskal

3.2.1 Algoritmo de solución

Aquí una simple representación de como funcionaría el algoritmo:

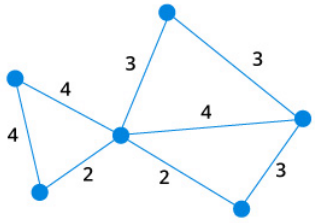


Figure 1: Comenzar con un grafo conexo.

Figure 2: Seleccionar la arista con menor peso.

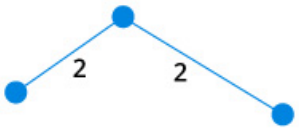


Figure 3: Seleccionar la siguiente arista con menos peso.

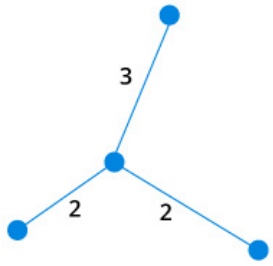


Figure 4: Seleccionar la siguiente arista con menos peso que no genere un ciclo.

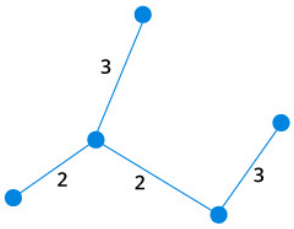


Figure 5: Seleccionar la siguiente arista con menos peso.

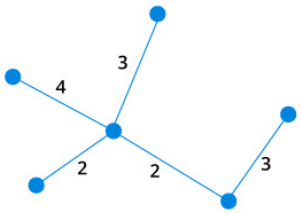


Figure 6: Repetir hasta obtener el árbol cobertor mínimo.

Para implementar el algoritmo como se describió anteriormente se hizo lo siguiente:

- Se utilizó Union-Find, con el fin de consultar si se generaría un ciclo.
- La representación del grafo conexo (Lista de adyacencia), se pasó a una cola de prioridad (Lista de aristas) para poder obtener de manera rápida las aristas de menor peso.
- Y gracias a lo anterior al retirar una arista, utilizando Find se verifica que no formen un ciclo, de no ser así, se agrega y se aplica Union para actualizar.

3.2.2 Implementación

```
   input: Adjacency List Adj
 1 T ← ∅
 2 weight ← 0
 3 UnionFind C ← {{v}, v ∈ Adj_V} //V = Vertex
 4 heapify(Adj_E)
 5 while |Adj_v| − 1 > |Adj_E| > 0 do
 6 |   Edge cur ← Adj_E.extractMin() // E = Edges
 7 |   if C.find(cur.u) ≠ C.find(cur.v) then
 8 |   |   T ← T ∪ cur
 9 |   |   weight ← weight + curr.weight
10 |   |   C.union(u, v)
11 |   end
12 end
13 return T
```

3.3 Dijkstra

3.3.1 Algoritmo de solución

Aquí una simple representación de como funcionaría el algoritmo:

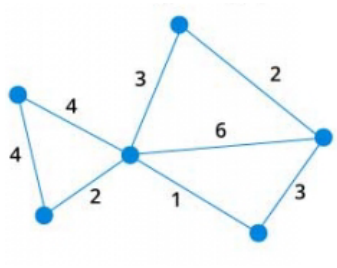


Figure 7: Comenzar con un grafo conexo.

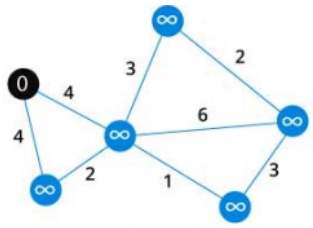


Figure 8: Escoger el nodo de inicio y asignar ∞ de distancia a los demás nodos.

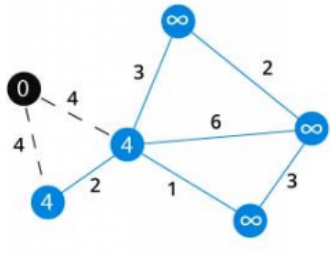


Figure 9: Ir a cada nodo adyacente del nodo y actualizar valores.

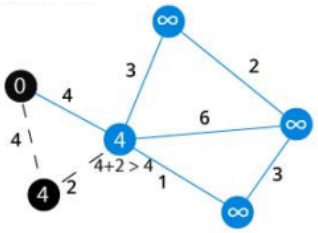


Figure 10: Si el camino del nodo adyacente es menor al nuevo camino, no se actualiza.

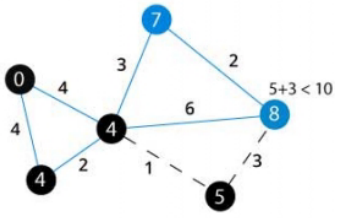


Figure 11: Después de cada iteración, se escoge el nodo no visitado con menos costo.

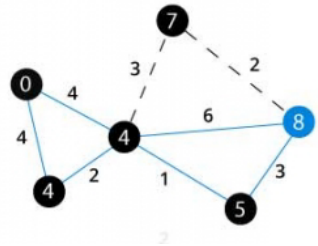


Figure 12: Repetir hasta haber visitado todos los nodos.

Para implementar el algoritmo como se describió anteriormente, se hizo lo siguiente:

- Se crearon dos arreglos, uno que contiene las distancias con ∞ y otro el camino hacia los nodos padre de cada nodo, éste nos ayuda a saber si ya fue visitado o no.
- Se creó una cola de prioridad donde se le fuerza la inserción de una tupla la cual contiene (peso , idNodo) ordenando de menor a mayor, esto ayuda a escoger los posibles caminos a visitar con menor peso.
- Si el camino del nodo adyacente es menor al nuevo camino, se actualizan los datos de distancia y padre, agregando esta nueva información también a la cola de prioridad.
- Repetir hasta que la cola de prioridad esté vacía.

3.3.2 Implementación

```
   input: Adjacency List Adj, Vertex s
 1 foreach v ∈ Adj_V do
 2 |   dist_v ← ∞
 3 |   dad_v ← NULL
 4 end
 5 Q ← {(0, s)}
 6 heapify(Q)
 7 while |Q| > 0 do
 8 |   d, here ← Q.extractMin()
 9 |   //Para cada edge que está conectado vertex here
10 |   foreach e ∈ ((Adj_V)_here)_E do
11 |   |   if dist_here + e.distance < dist_e.u then
12 |   |   |   dist_e.u ← dist_here + e.distance
13 |   |   |   dad_e.u ← here
14 |   |   |   Q ← Q ∪ (dist_e.u, e.u)
15 |   |   end
16 |   end
17 end
18 return dad
```

Para este pseudocódigo se muestra una versión que busca el camino más corto de un origen hacia todos los nodos, de igual forma se utilizó para obtener tiempos pero en la implementación se le agregó un nodo objetivo a buscar por defecto, en su implementación considera el nodo origen como el $nodo_0$ y el nodo destino como el $nodo_{n-1}$, en su opción verboso, se imprime el camino mínimo del destino al origen.

3.4 Modo de uso

Se incluyen 3 archivos:

- dijkstra.cpp
- kruskal.cpp
- utils.h

dijkstra.cpp y kruskal.cpp no requieren explicación, utils.h contiene funciones y estructuras necesarias para el correcto uso de los algoritmos, así que es importante que se encuentre al momento de su uso.

Para su ejecución es simple, tomemos como ejemplo kruskal, por línea de comando (UNIX) escribir lo siguiente:

$ g++ kruskal.cpp -o kruskal

$ ./kruskal 1000 1 1

La primera línea compilará el archivo y la segunda ejecutará kruskal con 1000 nodos , una densidad de 1 y se repetirá 1 vez la ejecución. también se puede ejecutar

$ ./kruskal [-v] 1000 1 1

Para que nos muestre el árbol cobertor mínimo.

Es muy importante destacar que la entrada debe de ser como se mencionó en el enunciado ya que no es validada.

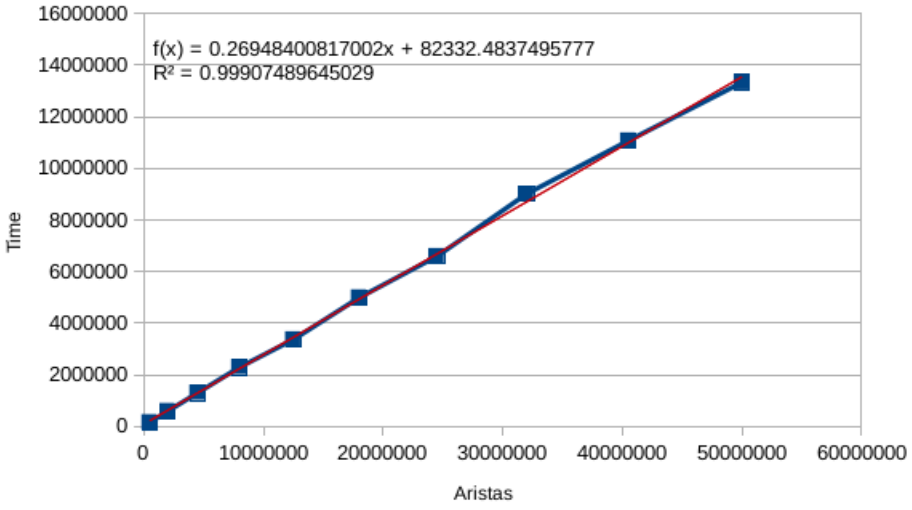
4 Pruebas

Se ejecutaron pruebas tal como se describió en el enunciado. Se adjunta un archivo excel que contiene todos los tiempos de ejecución, Desde la fila 1103 se encuentran los tiempos promedios para las cantidades de nodos y densidad. Es muy importante mencionar que estos tiempos fueron extraídos utilizando la medición CLOCKS PER SECECONS, pero en la implementación final se utilizan milisegundos.

Para una mejor facilidad, se utilizarán aquí simplemente los datos obtenidos en el promedio de ejecución con 10000 y densidad de 1.

4.1 Kruskal

Nodos	Aristas	Densidad	Tiempo
1000	499500	1	151592.6
2000	1999000	1	586408.1
3000	4498500	1	1297701.7
4000	7998000	1	2290014.3
5000	12497500	1	3363740.1
6000	17997000	1	4989436.1
7000	24496500	1	6593379.8
8000	31996000	1	9011528.9
9000	40495500	1	11074311.5
10000	49995000	1	13333472.5



```
vector<edge> kruskal(vector<vector<P>> adj)
{
    int n = adj.size();
    double weight = 0;
    int C[n], R[n];
    vector <edge> T;
    priority_queue <edge, vector <edge>> E;

    for(int i=0; i<n; ++i)// full anidated cilce O(E)
    {
        C[i] = i; R[i] = 0;
        for (int j = 0; j < adj[i].size(); ++j)
            E.push(edge(i,adj[i][j].second, adj[i][j].first)); //O(log(E))
    } //O(E*log(E))

    while(T.size() < n-1 && !E.empty()) //O(E) worst case, empty pq (O(E))
    {
        edge cur = E.top(); E.pop(); //(O(log(E)))
        int uc = find(C, cur.u), vc = find(C, cur.v); //alfa(v)
        if(uc != vc)
        {
            T.push_back(cur); weight += cur.d;
            if(R[uc] > R[vc]) C[vc] = uc;
            else if(R[vc] > R[uc]) C[uc] = vc;
            else { C[vc] = uc; R[uc]++; }
        }
    }
    return T;
} //O(E*(log(E) + log(v)))
```

Analizando el código podemos ver:

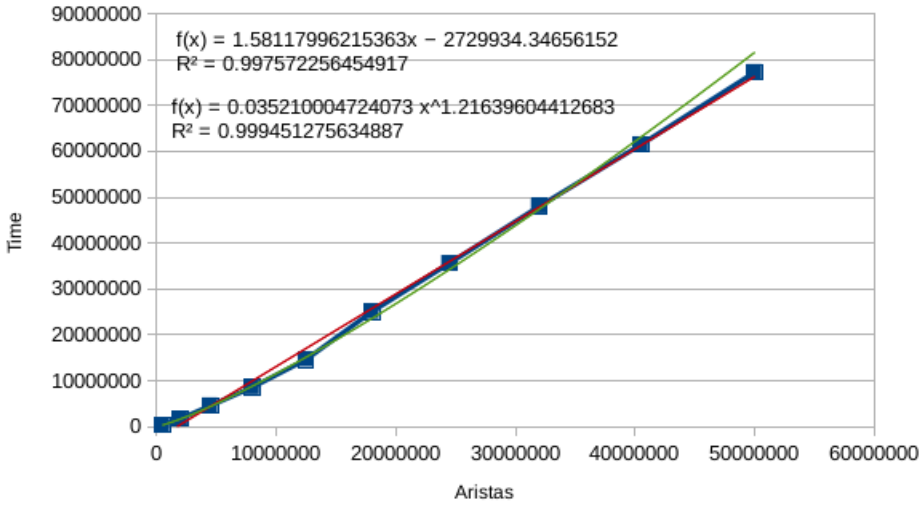
- La inicialización de la PQ y Unión-Find toma $\mathcal{O}(E \cdot log_2(E))$.
- La creación del MSD cuesta $\mathcal{O}(E \cdot (log_2(E) + \alpha(V))$.
- Entonces tenemos que nos cuesta $\mathcal{O}(2E \cdot log_2(E) + E \cdot \alpha(V))$.
- $\therefore \mathcal{O}(E \cdot log_2(E))$.

Si vemos nuestro gráfico tiene forma de $x \cdot log(x)$, pero no podemos afirmar nada sin antes analizarlo. Observando su linea de tendencia podemos observar:

- La tendencia roja es lineal y fácilmente va a ser una cota inferior.
- De aquí podemos decir que tenemos $\Theta(x)$.
- Como sabemos que nuestro tiempo teórico es $\mathcal{O}(x \cdot log_2(x))$ confirmemos que se cumpla el punto anterior.
- $\lim_{x\to\infty} \frac{x}{x \cdot log_2(x)} = 0 \therefore$ Se concluye que es $\mathcal{O}(x \cdot log_2(x))$.

4.2 Dijkstra

Nodos	Aristas	Densidad	Tiempo
1000	499500	1	288138.5
2000	1999000	1	1685811.8
3000	4498500	1	4526754.8
4000	7998000	1	8534128.3
5000	12497500	1	14592132.5
6000	17997000	1	25031167.5
7000	24496500	1	35609440.9
8000	31996000	1	48022693.7
9000	40495500	1	61493204.6
10000	49995000	1	77250844.2




```
vector<int> dijkstra(vector<vector<P>> edges, int s, int t)
{
    priority_queue<P, vector<P>, greater<P> > Q;
    vector<double> dist(edges.size(), INF); vector<int> dad(edges.size(), -1);
    Q.push (make_pair (0, s)); dist[s] = 0;
    while (!Q.empty())  //pq + for O(E+V)
    {
        P p = Q.top(); Q.pop();
        if (p.second == t) break;
        int here = p.second;
        for (int i = 0; i< edges[here].size(); ++i)
        {
            if (dist[here] + edges[here][i].first < dist[edges[here][i].second])
            {
                dist[edges[here][i].second] = dist[here] + edges[here][i].first;
                dad[edges[here][i].second] = here;
                Q.push (make_pair (dist[edges[here][i].second], edges[here][i].second
            }
        }
    } //O(E+V(Log(V)))
    return dad;
}
```

Analizando el código podemos ver:

- Realizar todo el ciclo anidado cuesta $\mathcal{O}(E + V)$.
- La extracción de la PQ cuesta $\mathcal{O}(log_2(V))$.
- Tenemos $\mathcal{O}((E + V) \cdot log_2(v))$.
- Como es un grafo conexo sabemos que $E \geq V - 1 \implies \mathcal{O}(2E \cdot log_2(v))$.
- $\therefore \mathcal{O}(E \cdot log_2(v))$.

Si vemos nuestro gráfico tiene forma de $x \cdot log(x)$ pero no podemos afirmar nada sin antes analizarlo y como sabemos que nuestro tiempo teórico es de $\mathcal{O}(E \cdot log_2(v))$ como no podemos analizar dos variables en nuestro gráfico y sabemos que $E \geq V - 1$ por simplicidad aceptaremos el error de decir que nuestra complejidad es de $\mathcal{O}(E \cdot log_2(E))$.

Observando su línea de tendencia podemos observar:

- La tendencia verde no es mayor que $x^{1.2}$ y fácilmente va a ser una cota superior.
- La tendencia roja es lineal y fácilmente va a ser una cota inferior.
- De aquí podemos decir que tenemos $\Theta(x) \wedge \Omega(x^{1.2})$.
- Como sabemos que nuestro tiempo teórico es $\mathcal{O}(x \cdot log_2(x))$ confirmemos que se cumpla el punto anterior.
- $\lim_{x\to\infty} \frac{x^{1.2}}{x \cdot log_2(x)} = \infty \wedge \lim_{x\to\infty} \frac{x}{x \cdot log_2(x)} = 0$.
- $\therefore$ Utilizando el teorema del sandwich se concluye que es $\mathcal{O}(x \cdot log_2(x))$.

5 Conclusión

Gracias a este trabajo, logré aprender bastante de grafos y cómo implementar sus algoritmos icónicos. También puse en práctica nuevas estructuras las cuales no había utilizado antes. Por último, también se obtuvieron los resultados esperados de este experimento sin ninguna mayor dificultad, salvo algunos errores menores que luego se corrigeron sin problemas.

6 Anexos

Grafos para búsqueda en espacios métricos Rodrigo Paredes Moraleda
https://users.dcc.uchile.cl/ raparede/publ/08PhDthesis.pdf

Como adjunto, también se encuentra un archivo excel que contiene más detalles sobre los tiempos del experimento.