

Por: Diego Alejandro Angarita Arboleda  
Momento práctico parcial 3

Objetivo del proyecto:

- Implementar y comparar BFS (mínimo número de aristas) y Dijkstra (pesos no negativos) como soluciones de SSSP (Single-Source Shortest Path..
- Justificar la aplicabilidad de cada algoritmo y sus complejidades.
- Diseñar E/S reproducible y medir tiempos.

Decisiones de diseño:

Lista de adyacencia sobre matriz de adyacencia:

Se eligió usar una lista de adyacencia sobre una matriz de adyacencia, debido a que estas son más rápidas en grafos dispersos, es decir, grafos donde el cuadrado de número de vértices es mayor que el número de aristas

Algoritmo de BFS basado en:

```
vector<int> bfs(vector<vector<int>>& adj) {  
    int V = adj.size();  
    vector<bool> visited(V, false);  
    vector<int> res;  
  
    queue<int> q;  
  
    int src = 0;  
    visited[src] = true;  
    q.push(src);  
  
    while (!q.empty()) {  
        int curr = q.front();  
        q.pop();  
        res.push_back(curr);  
  
        // visit all the unvisited  
        // neighbours of current node  
        for (int x : adj[curr]) {  
            if (!visited[x]) {  
                visited[x] = true;  
                q.push(x);  
            }  
        }  
    }  
  
    return res;  
}
```

Se hicieron los cambios necesarios para guardar el camino recorrido en un vector camino y otro vector que almacena los vértices padres.

Algoritmo de Dijkstra basado en:

```

vector<int> dijkstra(int V, vector<vector<int>>> &edges, int src){

    // Create adjacency list
    vector<vector<vector<int>>>> adj = constructAdj(edges, V);

    // Create a priority queue to store vertices that
    // are being preprocessed.
    priority_queue<vector<int>, vector<vector<int>>,
        greater<vector<int>>>> pq;

    // Create a vector for distances and initialize all
    // distances as infinite
    vector<int> dist(V, INT_MAX);

    // Insert source itself in priority queue and initialize
    // its distance as 0.
    pq.push({0, src});
    dist[src] = 0;

    // Looping till priority queue becomes empty (or all
    // distances are not finalized)
    while (!pq.empty()){

        // The first vertex in pair is the minimum distance
        // vertex, extract it from priority queue.
        int u = pq.top()[1];
        pq.pop();

        // Get all adjacent of u.
        for (auto x : adj[u]){

            // Get vertex label and weight of current
            // adjacent of u.
            int v = x[0];
            int weight = x[1];

            // If there is shorter path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}

```

Se hicieron los cambios necesarios para guardar el camino recorrido en un vector camino y otro vector que almacena los vértices padres. También se incluyó una parte para evitar que un vértice se repita más de una vez.

#### Algoritmo para leer y escribir archivos

Se implementaron las siguientes funciones:

Una función que lee los archivos, verificando que existan y que la información contenida en esta sea correcta.

Una función que escribe la información requerida requerida en la rúbrica en archivo csv.

Una función que lee los archivos y procesa cada caso para aplicar BFS o Dijkstra según se requiera.

Una función que escribe los resultados requeridos en un archivo txt.

Estructuras más usadas o de gran importancia:

vector: funciona como un arreglo dinámico para guardar una cantidad indeterminada de información.

queue(cola): también es un arreglo dinámico pero con una estructura FIFO(first in first out), por lo tanto, todas las operaciones se realizan sobre el primer elemento.

priority\_queue(cola de prioridad) min heap: es un tipo de cual en la cual se asegura que el elemento más pequeño, sea el primero de la lista, es útil para Dijkstra ya que nos permite manejar siempre la distancia (peso) más corta.

tuple(tuple): funciona igual que un vector pero la diferencia está en que una en que una tupla permite guardar múltiples valores, lo cual es útil para almacenar la información de las aristas (vertice, vertice, peso).

Problemas al desarrollar el proyecto:

-necesitar 2 vectores para reconstruir camino:

al momento de implementar la funcionalidades para imprimir el camino más corto, tuve problemas al insertar en el vector los vértices necesarios, pero al implementar otro vector que almacena los vértices padres, así que al usar los vértices padres para implementar para obtener el camino, nos aseguramos que el camino guardado es el más corto.

-necesitar 2 funciones para cada algoritmo:

al momento de implementar los algoritmos, entendí que la función que estaba implementando no iba a funcionar si el grafo era dirigido la misma función no iba a servir, tenía que crear una función que tuviera en cuenta la dirección de la arista, por lo tanto debía crear una nueva clase para grafos no dirigidos en la cual la union de vertices por una arista solo iban a funcionar de la forma (u,v,w) y no de la forma (v,u,w). Además también debía desarrollar una función para obtener los vecinos de un vértice, para que de esta forma los vértices sean vecinos de sus padres.

-error en las rutas de los archivos:

al momento de desarrollar los algoritmos para leer y escribir archivos nunca tome en cuenta que las carpetas data y results debían estar afuera de la carpeta src donde se almacenan los algoritmos, así cuando los algoritmos buscaban carpetas pero a no buscar fuera de src, aparecían como inexistentes, por lo cual al incluir la ruta a buscar, se debe añadir el prefijo “../”, para ir a la carpeta que contiene a src.

- "redefinición de la clase Grafo":

al momento de compilar el proyecto, aparecía un error que hacía referencia que la clase grafo se redefinía, esto pasaba porque al usar la clase grafo en todos los demás archivos, el archivo "Grafo" intentaba compilarse varias veces, por lo que hacían falta las siguientes líneas

```
#ifndef GRAFO_CPP
#define GRAFO_CPP

#endif
```

- problema al compilar:

siempre aparece un error de código referente a que no se está haciendo uso de la librería filesystem pero este error solo es por la versión del compilador y se soluciona, haciendo la compilación mediante el siguiente comando:  
g++ -std=c++17 main.cpp -o main -lstdc++fs (para compilador g++)

BFS vs Dijkstra

complejidad:

BFS:  $O(V + E)$ : esto es así ya que BFS recorre cada vértice solo un vez, y las aristas se recorren una o 2 veces dependiendo de si el grafo es dirigido o no.

Dijkstra:  $O((V + E) \log V)$ : en una min heap, extraer el vértice es de complejidad  $O(\log V)$ , este proceso se hace  $V$  veces, por lo tanto la complejidad es  $O(V \log V)$ , además, el proceso también se hace con cada arista, por lo que la complejidad total será  $O((V + E) \log V)$ .

aplicación:

BFS: minimiza la cantidad de saltos que se deben hacer entre para llegar de un vértice a otro.

Dijkstra: minimiza el costo para llegar de un vértice a otro tomando en cuenta los pesos (no pueden haber pesos negativos)

Casos de prueba:

grafo ponderado(con pesos no negativos)

Entrada

3	6	0	1
0	1	4	
1	0	4	
0	2	2	
2	0	2	
1	2	1	
2	1	1	

Salida

```
resultadosDIJKSTRA
Grafo: No dirigido
Vertices: n = 3
Aristas: m = 6
Fuente: s = 0
Tiempo: 0.063900 ms
distancias minimas desde el vertice 0:
Vertice 0: 0 | Camino: 0
Vertice 1: 3 | Camino: 0 → 2 → 1
Vertice 2: 2 | Camino: 0 → 2
```

grafo ponderado(con pesos negativos)

Entrada

4	5	0	1
0	1	5	
1	2	-3	
2	3	2	
0	3	10	
1	3	8	

Salida:

```

resultadosDIJKSTRA
Grafo: Dirigido
Vertices: n = 4
Aristas: m = 5
Fuente: s = 0
Tiempo: 0.000000 ms
ERROR: PESOS NEGATIVOS DETECTADOS

```

grafo no ponderado:

Entrada

4	4	0	0
0	1	1	
1	2	1	
2	3	1	
3	0	1	

Salida

```
resultadosBFS
Grafo: Dirigido
Vertices: n = 4
Aristas: m = 4
Fuente: s = 0
Tiempo: 0.037500 ms
distancias minimas desde el vertice 0:
Vertice  0:      0 | Camino: 0
Vertice  1:      1 | Camino: 0 → 1
Vertice  2:      2 | Camino: 0 → 1 → 2
Vertice  3:      3 | Camino: 0 → 1 → 2 → 3
```

grafo desconectado:

Entrada:

```
6 4 0 0
0 1 1
1 2 1
3 4 1
4 5 1
```

Salida:

```
resultadosBFS
Grafo: Dirigido
Vertices: n = 6
Aristas: m = 4
Fuente: s = 0
Tiempo: 0.042900 ms
distancias minimas desde el vertice 0:
Vertice  0:      0 | Camino: 0
Vertice  1:      1 | Camino: 0 → 1
Vertice  2: INF (inalcanzable)
Vertice  3: INF (inalcanzable)
Vertice  4: INF (inalcanzable)
Vertice  5: INF (inalcanzable)
```

archivo csv generado

```
caso,algoritmo,n,m,s,tipo,tiempo_ms
caso_desconectado,BFS,6,4,0,dirigido,0.042900
caso_no_ponderado,BFS,4,4,0,dirigido,0.041900
caso_pesos_negativos,Dijkstra,4,5,0,dirigido,0.000100
caso_ponderado,Dijkstra,3,6,0,no_dirigido,0.038300
```

### Bibliografía y referencias

Repositorio del profesor:

[https://github.com/carlosalvarezh/EstructuraDatosAlgoritmos1/blob/main/S08\\_EDNoLineales\\_Grafos.ipynb](https://github.com/carlosalvarezh/EstructuraDatosAlgoritmos1/blob/main/S08_EDNoLineales_Grafos.ipynb)

Algoritmo de BFS:

<https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/>

Algoritmo de Dijkstra:

<https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/>

### Usos de IA

IA: claude sonnet 4.5

prompts:

- Porque el programa dice que la librería filesystem no se usa cuando en realidad si la estoy usando.
- porque al reconstruir el camino, el vector camino no se muestra con los elementos correspondientes para Dijkstra y BFS
- Dame ejemplos de Grafos para usar como casos de pruebas.
- Como puedo medir tiempos de ejecución de forma efectiva en c++.