GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO DE DIJKSTRA

1. Introducción

En guías de aprendizaje anteriores especificamente en la que se abordó el recorrido a lo ancho (BFS) en los grafos se había dejado claro que cuando el grafo era no ponderado o la poderación de las aristas era la misma aplicando un BFS sobre el grafo partiendo de un nodo podiamos hallar el camino mínimo de ese nodo incial hacia todos los demás nodos del grafo.

Ahora como podríamos calcular el camino mínimo de un nodo hacia el resto de los nodos en un grafo donde la ponderación de las aristas a pesar de ser positiva no tiene que tener el mismo valor en cada una de las aristas.

Vamos a abordar el algortimo que resuelve este problema cuyo nombre es en honor de su creador Edgser Dijkstra quien lo describió por primera vez en 1959.

2. Conocimientos previos

2.1. Representación de grafos

Existen diferentes formas de representar un grafo. La estructura de datos usada depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

La estructura de datos usada depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

2.2. Cola con prioridad

Una cola de prioridad es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen, puede llegar a ofrecer la extracción constante de tiempo del elemento más grande (por defecto), a expensas de la inserción logarítmica, o utilizando greater<int>causaría que el menor elemento aparezca como la parte superior con .top(). Trabajar con una cola de prioridad es similar a la gestión de un heap

3. Desarrollo

De lo que este algoritmo trata es de marcar todos los vértices como no utilizados. El algoritmo parte de un vértice origen que será ingresado, a partir de ese vértices evaluaremos sus adyacentes, como dijkstra usa una técnica greedy - La técnica greedy utiliza el principio de que para que un

camino sea óptimo, todos los caminos que contiene también deben ser óptimos- entre todos los vértices adyacentes, buscamos el que esté más cerca de nuestro punto origen, lo tomamos como punto intermedio y vemos si podemos llegar más rápido a través de este vértice a los demás. Después escogemos al siguiente más cercano (con las distancias ya actualizadas) y repetimos el proceso. Esto lo hacemos hasta que el vértice no utilizado más cercano sea nuestro destino. Al proceso de actualizar las distancias tomando como punto intermedio al nuevo vértice se le conoce como relajación.

Teniendo un grafo dirigido ponderado de N nodos no aislados, sea x el nodo inicial, un vector D de tamaño N guardará al final del algoritmo las distancias desde x al resto de los nodos.

1. Inicializar todas las distancias en D con un valor infinito relativo ya que son desconocidas al principio, exceptuando la de x que se debe colocar en 0 debido a que la distancia de x a x sería 0.
2. Sea a = x (tomamos a como nodo actual).
3. Recorremos todos los nodos adyacentes de a, excepto los nodos marcados, llamaremos a estos nodos no marcados v_i.
4. Para el nodo actual, calculamos la distancia tentativa desde dicho nodo a sus vecinos con la siguiente fórmula: dt(v_i.) = D_a. + d(a,v_i.). Es decir, la distancia tentativa del nodo 'v_i.'es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho el nodo 'a'(el actual) al nodo v_i. Si la distancia tentativa es menor que la distancia almacenada en el vector, actualizamos el vector con esta distancia tentativa. Es decir: Si dt(v_i) $< D_{vi} \rightarrow D_{vi}$ = dt(v_i)
5. Marcamos como completo el nodo a.
6. Tomamos como próximo nodo actual el de menor valor en D (puede hacerse almacenando los valores en una cola de prioridad) y volvemos al paso 3 mientras existan nodos no marcados.

Una vez terminado al algoritmo, D estará completamente lleno.

4. Implementación

Veamos la implementación de este algoritmo en ambos lenguajes

4.1. C++

```
#include <vector>
#include <algorithm>
#include <queue>
#include <stack>
#define MAX_N 100001
#define INF 987654321
using namespace std;
```

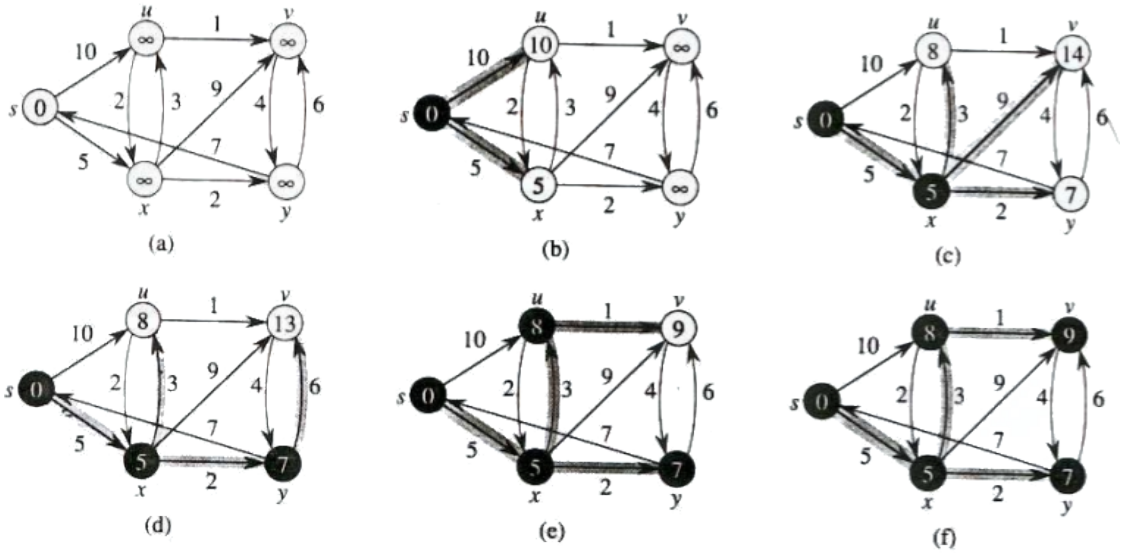

Figura 1: Ejecución del algoritmo Dijkstra comenzando por el nodo *s*.

```
int nnodes;

struct Node{
   int dist;
   vector<int> adj;
   vector<int> weight;
};
Node graf[MAX_N];
bool mark[MAX_N];

struct pq_entry{
   int node, dist;
   bool operator <(const pq_entry &a) const{
      if (dist != a.dist) return (dist > a.dist);
         return (node > a.node);
   }
};

inline void dijkstra(int source){
   priority_queue<pq_entry> pq;
   pq_entry P;
   for (int i=0;i<nnodes;i++){
      if (i == source){
         graf[i].dist = 0;
         P.node = i;
         P.dist = 0;
         pq.push(P);
```

```
        }
        else graf[i].dist = INF;
    }
    while (!pq.empty()){
        pq_entry curr = pq.top();
        pq.pop();
        int nod = curr.node;
        int dis = curr.dist;
        for(int i=0;i<graf[nod].adj.size();i++){
            if(!mark[graf[nod].adj[i]]){
                int nextNode = graf[nod].adj[i];
                if(dis + graf[nod].weight[i] < graf[nextNode].dist){
                    graf[nextNode].dist = dis + graf[nod].weight[i];
                    P.node = nextNode;
                    P.dist = graf[nextNode].dist;
                    pq.push(P);
                }
            }
        }
        mark[nod] = true;
    }
}

int main(){
    n = 4;
    graf[0].adj.push_back(1);
    graf[0].weight.push_back(5);
    graf[1].adj.push_back(0);
    graf[1].weight.push_back(5);

    graf[1].adj.push_back(2);
    graf[1].weight.push_back(5);
    graf[2].adj.push_back(1);
    graf[2].weight.push_back(5);

    graf[2].adj.push_back(3);
    graf[2].weight.push_back(5);
    graf[3].adj.push_back(2);
    graf[3].weight.push_back(5);

    graf[3].adj.push_back(1);
    graf[3].weight.push_back(6);
    graf[1].adj.push_back(3);
    graf[1].weight.push_back(6);

    dijkstra(0);

    cout<<graf[3].dist<<endl;
    return 0;
}
```

4.2. Java

```java
public class Main {

   public static void shortestPaths(List<Edge>[] edges, int s, int[] prio, int
       [] pred){
      Arrays.fill(pred, -1);
      Arrays.fill(prio, Integer.MAX_VALUE);
      prio[s] = 0;
      PriorityQueue<Long> q = new PriorityQueue<>();
      q.add((long) s);
      while (!q.isEmpty()) {
        long cur = q.remove();
        int curu = (int) cur;
        if (cur >>> 32 != prio[curu])
        continue;
        for(Edge e : edges[curu]){
           int v = e.t;
           int nprio = prio[curu]+e.cost;
           if(prio[v] > nprio){
              prio[v] = nprio;
              pred[v] = curu;
              q.add(((long) nprio << 32) + v);
           }
        }
      }
   }

   static class Edge{
      int t, cost;
      public Edge(int t, int cost){
         this.t = t;
         this.cost = cost;
      }
   }

   // Ejemplo de uso
   public static void main(String[] args) {
      int[][] cost = { { 0, 3, 2 }, { 0, 0, -2 }, { 0, 0, 0 } };
      int n = cost.length;
      List<Edge>[] edges = new List[n];
      for(int i = 0; i < n; i++) {
         edges[i] = new ArrayList<>();
         for(int j = 0; j < n; j++) {
            if(cost[i][j] != 0) {
               edges[i].add(new Edge(j, cost[i][j]));
            }
```

```
            }
        }
        int[] dist = new int[n];
        int[] pred = new int[n];
        shortestPaths(edges, 0, dist, pred);
    }
}
```

En ambas implementaciones se asume que el grafo en ponderado no dirigido.

5. Complejidad

La complejidad del algoritmo O($|V|^2$) sin utilizar cola de prioridad, O((|A|+|V|) log |V|) = O(|A| log |V|) utilizando cola de prioridad (por ejemplo un montículo). Por otro lado, si se utiliza un Montículo de Fibonacci, sería O(|V| log |V|+|A|).

Donde V es la cantidad de vértices y A las aristas del grafo. En la implentaciones anteriores tanto la de C++ y Java son O($(V + A)logV$).

6. Aplicaciones

El problema del camino más corto es clásico en los grafos, como el nombre lo indica se trata de hallar el camino más corto desde un nodo hacia todos los demás. El algoritmo de Dijkstra resuelve esto sobre un grafo donde la ponderación de las aristas sea un valor igual o mayor que cero nunca menor que este porque el mismo algoritmo caería en un bucle infito.

El algoritmo Dijkstra es un ejemplo del algoritmo greedy tomando la mejor opción entre ir directo o ir con algunos nodos intermedios.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando este algoritmo:

- DMOJ - Red de Tranvías
- DMOJ - Transferencia Electrónica
- DMOJ - Fiesta Vacuna
- DMOJ - Alex y su país
- DMOJ - Dragones de IslaGrande
- DMOJ - Otra fiesta vacuna
- DMOJ - Feria tecnológica