Algoritmul Dijkstra

Algoritmul **Dijkstra** este utilizat pentru a găsi cele mai scurte drumuri de la un nod sursă la toate celelalte noduri într-un graf ponderat. Acest algoritm utilizează o coadă de priorități pentru a selecta nodul cu distanța minimă într-un mod eficient și pentru a actualiza distanțele vecinilor săi. Este un algoritm destul de eficient și se aplică, în general, în grafuri cu greutăți pozitive.

Așa cum am menționat, algoritmul pornește de la un nod sură care va avea distanța 0. Într-o forma simplistă putem urmări in 3 etape cum funcționează algoritmul Dijkstra.

1. **Inițializare:** 
   * Se definește un vector dist[] care va conține distanțele minime de la nodul sursă la fiecare nod din graf unde inițial toate valorile cu excepția nodului sursa, sunt setate la infinit. Se creează o coada de priorități (sau un set de noduri neexplorate) pentru a alege nodul cu distanța minimă neexplorată.
2. **Etapa de procesare:** 
   * Se extrage nodul cu distanța minimă din coada de priorități și se actualizează distanțele pentru vecinii nodului curent. Dacă distanța de la nodul curent la vecinul respectiv este mai mică decât valoarea deja stocată în vectorul dist[], se actualizează distanța pentru acel vecin și se adaugă vecinii actualizați in coada de priorități pentru a-i procesa ulterior.
3. **Finalizare**
   * Se repetă acest proces până când toate nodurile din graf au fost procesate/ La final, vectorul dist[] va conține cele mai scurte distanțe de la nodul sură la fiecare nod din graf.

Complexitatea algoritmului este de *O((V+E)* log*V)* pentru timp, unde V este numarul de noduri si E numărul de muchii ale grafului. Pentru spațiu, prezintă o complexitate de *O(V+E)* datorita stocării grafului si a distanțelor.

# Reprezentarea algoritmului Dijkstra în C++:

Dijkstra.cpp

#include <iostream>

#include <vector>

#include <queue>

#include <climits> // pentru a folosi valoarea INT\_MAX

#include <algorithm> // pentru reverse()

using namespace std;

// Definim un tip pentru stocarea perechilor (costul muchiei, nodul vecin)

typedef pair<int, int> pii;

class Graph {

public:

int V; // nr de noduri din graf

vector<vector<pii>> adj; // lista de adiacență (stocăm muchiile și greutățile lor)

// Constructor

Graph(int V) {

this->V = V;

adj.resize(V); // alocăm memoria pentru lista de adiacență

}

// Funcție pentru a adăuga o muchie între nodul u(curent) și nodul v(vecin), cu greutatea weight

void addEdge(int u, int v, int weight) {

adj[u].push\_back({ weight, v }); // adăugăm muchia (u, v) cu greutatea

adj[v].push\_back({ weight, u }); // muchia inversă (v, u), pentru graf neorientat

}

// Algoritmul Dijkstra

void dijkstra(int start) {

// Vector de distanțe pentru fiecare nod

vector<int> dist(V, INT\_MAX);

vector<int> parent(V, -1);

dist[start] = 0;

// Coada de priorități pentru a selecta nodul cu distanța minimă

priority\_queue<pii, vector<pii>, greater<pii>> pq; //Coada de tip min-heap

pq.push({ 0, start }); //nodul sursă în coada de priorități cu distanța 0

while (!pq.empty()) {

int u = pq.top().second; // Extragem nodul cu distanța minimă

pq.pop(); // Îl eliminăm din coada de priorități

// Parcurgem toți vecinii nodului curent

for (auto& edge : adj[u]) {

int weight = edge.first; // Greutatea muchiei (distanta dintre u și v)

int v = edge.second; // Nodul vecin v

// actualizare daca un drum mai scurt e gasit.

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

parent[v] = u;

pq.push({ dist[v], v }); // v(vecinul) in coada de prioritati

}

}

}

// Afișăm rezultatele

cout << "Nod\tDistanta\tDrum\n";

for (int i = 0; i < V; i++) {

if (dist[i] == INT\_MAX) {

cout << i << "\tINF\t\tNu exista drum\n";

}

else {

cout << i << "\t" << dist[i] << "\t\t";

vector<int> path;

for (int v = i; v != -1; v = parent[v])

path.push\_back(v);

reverse(path.begin(), path.end());

for (int v : path)

cout << v << " ";

cout << endl;

}

}

}

};

int main() {

Graph g(5);

g.addEdge(0, 1, 10); // Nodul 0 este conectat la nodul 1 cu greutatea 10

g.addEdge(0, 2, 5); // Nodul 0 este conectat la nodul 2 cu greutatea 5

g.addEdge(1, 2, 2); // Nodul 1 este conectat la nodul 2 cu greutatea 2

g.addEdge(1, 3, 1); // Nodul 1 este conectat la nodul 3 cu greutatea 1

g.addEdge(2, 3, 9); // Nodul 2 este conectat la nodul 3 cu greutatea 9

g.addEdge(2, 4, 2); // Nodul 2 este conectat la nodul 4 cu greutatea 2

g.addEdge(3, 4, 4); // Nodul 3 este conectat la nodul 4 cu greutatea 4

g.dijkstra(0);

return 0;

}

Output pentru cel mai mic cost de la nodul 0 spre toate nodurile afișând si drumul:

10

5

2

4

9

1

2

