Algoritmul de Cautare Bruta

Algoritmul de cautare bruta este o metoda de cautare exhaustiva care exploreaza toate caile posibile intr-un graf pentru a gasi drumul optim intre doua noduri. Desi nu este la fel de eficient ca Dijkstra sau A\*, acest algoritm garanteaza gasirea celui mai scurt drum prin explorarea tuturor posibilitatilor.

Descriere

Algoritmul functioneaza prin explorarea recursiva a tuturor cailor posibile din graf, folosind tehnica de backtracking. Pentru fiecare cale explorata, algoritmul tine evidenta costului total si actualizeaza solutia optima atunci cand gaseste un drum mai bun.

Pasi principali

1. Initializare:

   - Se marcheaza toate nodurile ca nevizitate

   - Se initializeaza costul minim cu infinit

   - Se pregateste vectorul pentru drumul curent

2. Explorare recursiva:

   - Se marcheaza nodul curent ca vizitat

   - Se adauga nodul la drumul curent

   - Se verifica daca s-a ajuns la destinatie

   - Se exploreaza recursiv toti vecinii nevizitati

3. Backtracking:

   - Se elimina nodul curent din drum

   - Se marcheaza nodul ca nevizitat

   - Se continua cu alte cai posibile

Avantaje

- Garanteaza gasirea celui mai scurt drum

- Simplu de inteles si implementat

- Nu necesita euristici sau structuri de date complexe

Dezavantaje

- Complexitate temporala foarte mare

- Ineficient pentru grafuri mari

- Timp de executie creste exponential cu numarul de noduri

Aplicatii practice

Implementarea foloseste aceeasi structura de date ca algoritmii Dijkstra si A\*, permitand o comparatie directa a performantei.

#include <iostream>

#include <vector>

#include <climits>

#include <algorithm>

using namespace std;

typedef pair<int, int> pii;

class Graph {

public:

int V; // nr de noduri din graf

vector<vector<pii>> adj; // lista de adiacenta

// Constructor

Graph(int V) {

this->V = V;

adj.resize(V);

}

// Functie pentru a adauga o muchie intre noduri

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

adj[u].push\_back({ weight, v });

adj[v].push\_back({ weight, u });

}

// Functia principala pentru cautarea bruta

void bruteForceSearch(int start, int end) {

vector<bool> visited(V, false);

vector<int> currentPath;

vector<int> bestPath;

int minCost = INT\_MAX;

// Pornim cautarea recursiva

searchPath(start, end, visited, currentPath, 0, minCost, bestPath);

// Afisam rezultatul

if (minCost == INT\_MAX) {

cout << "Nu exista drum intre nodurile " << start << " si " << end << endl;

}

else {

cout << "Cel mai scurt drum gasit:\n";

cout << "Cost total: " << minCost << "\n";

cout << "Drum: ";

for (int node : bestPath) {

cout << node << " ";

}

cout << endl;

}

}

private:

void searchPath(int current, int end, vector<bool>& visited,

vector<int>& currentPath, int currentCost,

int& minCost, vector<int>& bestPath) {

// Marcam nodul curent ca vizitat

visited[current] = true;

currentPath.push\_back(current);

// Daca am ajuns la destinatie, verificam daca am gasit un drum mai bun

if (current == end) {

if (currentCost < minCost) {

minCost = currentCost;

bestPath = currentPath;

}

}

else {

// Exploram toti vecinii nevizitati

for (const auto& edge : adj[current]) {

int weight = edge.first;

int neighbor = edge.second;

if (!visited[neighbor] && currentCost + weight < minCost) {

searchPath(neighbor, end, visited, currentPath,

currentCost + weight, minCost, bestPath);

}

}

}

// Backtracking

visited[current] = false;

currentPath.pop\_back();

}

};

int main() {

Graph g(5);

// Adaugam muchiile cu costurile lor

g.addEdge(0, 1, 10);

g.addEdge(0, 2, 5);

g.addEdge(1, 2, 2);

g.addEdge(1, 3, 1);

g.addEdge(2, 3, 9);

g.addEdge(2, 4, 2);

g.addEdge(3, 4, 4);

// Cautam drumul de la nodul 0 la nodul 4

g.bruteForceSearch(0, 4);

return 0;

}