Algoritmul A\*

Algoritmul A\* este o metodă de căutare informată care găsește cel mai scurt drum între două noduri într-un graf. Este o îmbunătățire a algoritmului Dijkstra, folosind o euristică pentru a ghida căutarea către țintă. A\* combină informații despre costul real al drumului parcurs (g(n)) cu o estimare euristică a distanței rămase până la țintă.

Pașii principali ai algoritmului A\*:

1. Inițializare:

- Se definesc vectorii g\_score și f\_score pentru a ține evidența costurilor

- Se inițializează nodul de start cu g\_score = 0 și f\_score = h(start)

- Se adaugă nodul de start în coada de priorități

2. Etapa de procesare:

- Se extrage nodul cu cel mai mic f\_score din coada de priorități

- Pentru fiecare vecin al nodului curent:

\* Se calculează noul g\_score

\* Dacă noul g\_score este mai mic decât cel existent, se actualizează

\* Se actualizează f\_score = g\_score + h(vecin)

\* Se adaugă vecinul în coada de priorități

3. Finalizare:

- Algoritmul se oprește când găsește nodul țintă sau când coada devine goală

Complexitatea algoritmului este O((V+E) log V) pentru timp, unde V este numărul de noduri și E numărul de muchii ale grafului. Pentru spațiu, prezintă o complexitate de O(V+E).

Reprezentarea algoritmului A\* în C++:

// A\_Algoritm.cpp : This file contains the 'main' function. Program execution begins and ends there.

//

#include <iostream>

#include <vector>

#include <queue>

#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

vector<pair<int, int>> coordinates; // coordonatele nodurilor pentru euristica

// Constructor

Graph(int V) {

this->V = V;

adj.resize(V);

coordinates.resize(V);

}

// Seteaza coordonatele pentru un nod (folosite în calculul euristic)

void setNodeCoordinates(int node, int x, int y) {

coordinates[node] = {x, y};

}

// Adauga o muchie între noduri

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

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

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

}

// Funcția euristica (distanța Manhattan)

int heuristic(int node, int goal) {

return abs(coordinates[goal].first - coordinates[node].first) +

abs(coordinates[goal].second - coordinates[node].second);

}

// Algoritmul A\*

void astar(int start, int goal) {

vector<int> g\_score(V, INT\_MAX); // costul real pana la nod

vector<int> f\_score(V, INT\_MAX); // scorul total (g + h)

vector<int> parent(V, -1); // pentru reconstructia drumului

g\_score[start] = 0;

f\_score[start] = heuristic(start, goal);

// Coada de prioritati pentru nodurile de explorat

priority\_queue<pii, vector<pii>, greater<pii>> pq;

pq.push({f\_score[start], start});

while (!pq.empty()) {

int current = pq.top().second;

pq.pop();

if (current == goal) {

// Am găsit destinatia, afisam rezultatul

printPath(start, goal, parent, g\_score);

return;

}

// Explorăm vecinii nodului curent

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

int weight = edge.first; // costul muchiei

int neighbor = edge.second; // nodul vecin

int tentative\_g\_score = g\_score[current] + weight;

if (tentative\_g\_score < g\_score[neighbor]) {

parent[neighbor] = current;

g\_score[neighbor] = tentative\_g\_score;

f\_score[neighbor] = g\_score[neighbor] + heuristic(neighbor, goal);

pq.push({f\_score[neighbor], neighbor});

}

}

}

cout << "Nu exista drum intre nodul " << start << " si nodul " << goal << endl;

}

private:

void printPath(int start, int goal, const vector<int>& parent, const vector<int>& g\_score) {

vector<int> path;

for (int current = goal; current != -1; current = parent[current]) {

path.push\_back(current);

}

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

cout << "Drumul gasit de la " << start << " la nodul " << goal << ":\n";

cout << "Costul total este:" << g\_score[goal] << "\n";

cout << "Drumul identificat este:";

for (int node : path) {

cout << node << " ";

}

cout << "\n";

}

};

int main() {

Graph g(5);

// Setam coordonatele nodurilor pentru calculul euristic

g.setNodeCoordinates(0, 0, 0);

g.setNodeCoordinates(1, 2, 2);

g.setNodeCoordinates(2, 3, 1);

g.setNodeCoordinates(3, 5, 3);

g.setNodeCoordinates(4, 4, 4);

// 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);

// Gasim drumul de la nodul 0 la nodul 4

g.astar(0, 4);

return 0;

}

Diferențe principale față de Dijkstra:

1. A\* folosește o euristică pentru a ghida căutarea către țintă

2. A\* este mai eficient pentru găsirea unui drum specific între două noduri

3. A\* garantează găsirea celui mai scurt drum doar dacă euristica este admisibilă