**AI Assignment 3**

Sharvari Deshmukh

TY CSA 73

**Problem Statement**-Implementation of Informed strategies

**Theory-**

**Best-First Search:**

**- Objective**: Best-First Search is an informed search algorithm used in artificial intelligence and graph traversal. It aims to find the optimal path to a goal state based on a heuristic function.

**- Heuristic Function**: The algorithm employs a heuristic function to estimate the cost or distance from the current state to the goal. This heuristic guides the search by selecting the most promising path first.

**- Priority Queue**: Nodes in the search space are stored in a priority queue, where the priority is determined by the heuristic function. Nodes with lower heuristic values are explored first.

**- Process**: The algorithm iteratively selects and expands the node with the lowest heuristic value from the priority queue. This process continues until the goal state is reached or the entire search space is explored.

**- Completeness and Optimality**: Best-First Search is not guaranteed to be complete or optimal. The quality of the solution depends on the effectiveness of the heuristic function. In some cases, it may find a solution faster than uninformed search algorithms, but it may not always guarantee the best possible solution.

**- Applications**: Best-First Search is commonly used in pathfinding, game playing, and various AI applications where an informed search strategy is beneficial.

**A\* Algorithm:**

**- Objective:** A\* is a widely used informed search algorithm designed for finding the shortest path or optimal solution from a start state to a goal state in a graph.

- **Heuristic Function:** Similar to Best-First Search, A\* employs a heuristic function to estimate the cost from the current state to the goal. The heuristic guides the search by prioritizing nodes with a combination of the actual cost incurred from the start state and the estimated cost to reach the goal.

- **Cost Function:**  A\* evaluates nodes based on a cost function, which is the sum of the actual cost to reach the node from the start state (denoted as g(n)) and the estimated cost from the node to the goal (denoted as h(n)).

**- Priority Queue:** Nodes are stored in a priority queue, sorted by their total cost (f(n) = g(n) + h(n)). Nodes with lower total costs are explored first.

- **Optimality:** A\* is both complete and optimal under certain conditions. It is guaranteed to find the optimal solution if the heuristic function is admissible (never overestimates the true cost) and consistent (satisfies the triangle inequality).

- **Applications:** A\* is widely used in pathfinding, robotics, and AI applications where finding the optimal path is crucial. It is known for its efficiency in exploring the search space.

*Best-First Search is an informed search algorithm guided by a heuristic function to prioritize nodes based solely on the estimated cost to the goal. A\* algorithm, an extension of Best-First Search, incorporates both the actual cost from the start state and the heuristic estimate, making it complete and optimal under certain conditions.*

**Comparison between A\* and BFS-**

| **Feature** | **A\* Search** | **Best-First Search** |
| --- | --- | --- |
| **Optimality** | Guarantees optimality if heuristic is admissible | Does not guarantee optimality |
| **Completeness** | Complete if finite search space and branching | Not necessarily complete |
| **Time Complexity** | Exponential, but can be near-linear with good heuristic | Exponential in worst case |
| **Memory Usage** | Typically requires more memory due to open and closed lists | May require less memory as it only maintains open list |
| **Applications** | Pathfinding, route planning, robotics | Web search engines, scheduling, pattern recognition |

1. **Romania Problem**

* **Goal:** To determine the optimal or most efficient path between a starting point and a destination within a given environment, considering factors such as distance, cost, or time.
* **Top of Form**
* **CODE USING BEST FIRST SEARCH–**

#include <iostream>

#include <vector>

#include <map>

#include <queue>

#include <string>

#include <algorithm>

using namespace std;

// Define the cities and their connections

map<string, map<string, int>> graph = {

    {"Arad", {{"Zerind", 75}, {"Timisoara", 118}, {"Sibiu", 140}}},

    {"Zerind", {{"Arad", 75}, {"Oradea", 71}}},

    {"Oradea", {{"Zerind", 71}, {"Sibiu", 151}}},

    {"Timisoara", {{"Arad", 118}, {"Lugoj", 111}}},

    {"Lugoj", {{"Timisoara", 111}, {"Mehadia", 70}}},

    {"Mehadia", {{"Lugoj", 70}, {"Drobeta", 75}}},

    {"Drobeta", {{"Mehadia", 75}, {"Craiova", 120}}},

    {"Craiova", {{"Drobeta", 120}, {"Pitesti", 138}, {"Rimnicu Vilcea", 146}}},

    {"Rimnicu Vilcea", {{"Craiova", 146}, {"Sibiu", 80}, {"Pitesti", 97}}},

    {"Sibiu", {{"Arad", 140}, {"Oradea", 151}, {"Rimnicu Vilcea", 80}, {"Fagaras", 99}}},

    {"Fagaras", {{"Sibiu", 99}, {"Bucharest", 211}}},

    {"Pitesti", {{"Rimnicu Vilcea", 97}, {"Craiova", 138}, {"Bucharest", 101}}},

    {"Bucharest", {{"Fagaras", 211}, {"Pitesti", 101}}},

};

// Heuristic function (straight-line distance to Bucharest)

map<string, int> heuristic = {

    {"Arad", 366}, {"Zerind", 374}, {"Oradea", 380}, {"Timisoara", 329},

    {"Lugoj", 244}, {"Mehadia", 241}, {"Drobeta", 242}, {"Craiova", 160},

    {"Rimnicu Vilcea", 193}, {"Sibiu", 253}, {"Fagaras", 178}, {"Pitesti", 98},

    {"Bucharest", 0}

};

// Best First Search function

vector<string> bestFirstSearch(const string &start, const string &goal) {

    priority\_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int, string>>> frontier;

    //element with the smallest heuristic value (closest to the goal) will be at the top of the priority queue.

    frontier.push({heuristic[start], start});

    map<string, string> cameFrom; // represents a city and the city from which it was reached

    cameFrom[start] = "";

    while (!frontier.empty()) {

        string current = frontier.top().second;

        frontier.pop();

        if (current == goal) {

            vector<string> path;

            while (current != "") {

                path.push\_back(current);

                current = cameFrom[current];

            }

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

            return path;

        }

        for (const auto &neighbor : graph[current]) { //explore neighbors of current by iterating over the adjacent cities stored in graph map

            string next = neighbor.first;

            if (cameFrom.find(next) == cameFrom.end()) { //if next has not been visited yet (not present in cameFrom map)

                cameFrom[next] = current; //stores current as the previous city

                int priority = heuristic[next]; // calculate priority (heuristic value) of next using the heuristic map

                frontier.push({priority, next}); //pushes {priority, next} onto the frontier queue

            }

        }

    }

    return {}; // no path found

}

int main() {

    string start = "Arad";

    string goal = "Bucharest";

    vector<string> path = bestFirstSearch(start, goal);

    if (!path.empty()) {

        cout << "Shortest path from " << start << " to " << goal << ":\n";

        for (const string &city : path) {

            cout << city << " -> ";

        }

        cout << "Goal reached.\n";

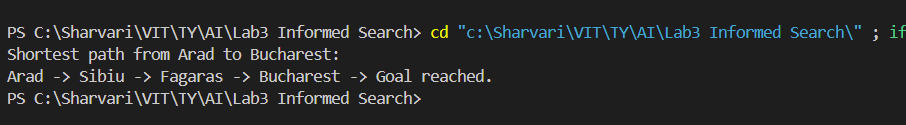
    } else {

        cout << "No path found from " << start << " to " << goal << ".\n";

    }

    return 0;

}



* **CODE FOR A\*–**

#include <iostream>

#include <vector>

#include <map>

#include <queue>

#include <string>

#include <algorithm>

#include <functional>

using namespace std;

// Define the cities and their connections

map<string, map<string, int>> graph = {

    {"Arad", {{"Zerind", 75}, {"Timisoara", 118}, {"Sibiu", 140}}},

    {"Zerind", {{"Arad", 75}, {"Oradea", 71}}},

    {"Oradea", {{"Zerind", 71}, {"Sibiu", 151}}},

    {"Timisoara", {{"Arad", 118}, {"Lugoj", 111}}},

    {"Lugoj", {{"Timisoara", 111}, {"Mehadia", 70}}},

    {"Mehadia", {{"Lugoj", 70}, {"Drobeta", 75}}},

    {"Drobeta", {{"Mehadia", 75}, {"Craiova", 120}}},

    {"Craiova", {{"Drobeta", 120}, {"Pitesti", 138}, {"Rimnicu Vilcea", 146}}},

    {"Rimnicu Vilcea", {{"Craiova", 146}, {"Sibiu", 80}, {"Pitesti", 97}}},

    {"Sibiu", {{"Arad", 140}, {"Oradea", 151}, {"Rimnicu Vilcea", 80}, {"Fagaras", 99}}},

    {"Fagaras", {{"Sibiu", 99}, {"Bucharest", 211}}},

    {"Pitesti", {{"Rimnicu Vilcea", 97}, {"Craiova", 138}, {"Bucharest", 101}}},

    {"Bucharest", {{"Fagaras", 211}, {"Pitesti", 101}}},

};

// Heuristic function (straight-line distance to Bucharest)

map<string, int> heuristic = {

    {"Arad", 366}, {"Zerind", 374}, {"Oradea", 380}, {"Timisoara", 329},

    {"Lugoj", 244}, {"Mehadia", 241}, {"Drobeta", 242}, {"Craiova", 160},

    {"Rimnicu Vilcea", 193}, {"Sibiu", 253}, {"Fagaras", 178}, {"Pitesti", 98},

    {"Bucharest", 0}

};

// A\* Search function

vector<string> aStarSearch(const string &start, const string &goal) {

    priority\_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int, string>>> frontier;

    frontier.push({heuristic[start], start});

    map<string, string> cameFrom;

    map<string, int> costSoFar;

    cameFrom[start] = "";

    costSoFar[start] = 0;

    while (!frontier.empty()) {

        string current = frontier.top().second;

        frontier.pop();

        if (current == goal) {

            vector<string> path;

            while (current != "") {

                path.push\_back(current);

                current = cameFrom[current];

            }

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

            return path;

        }

        for (const auto &neighbor : graph[current]) {

            string next = neighbor.first;

            int newCost = costSoFar[current] + neighbor.second;

            if (costSoFar.find(next) == costSoFar.end() || newCost < costSoFar[next]) {

                costSoFar[next] = newCost;

                cameFrom[next] = current;

                int priority = newCost + heuristic[next];

                frontier.push({priority, next});

            }

        }

    }

    return {}; // No path found

}

int main() {

    string start = "Arad";

    string goal = "Bucharest";

    vector<string> path = aStarSearch(start, goal);

    if (!path.empty()) {

        cout << "Shortest path from " << start << " to " << goal << ":\n";

        for (const string &city : path) {

            cout << city;

            if (city != goal) {

                cout << " -> ";

            }

        }

        cout << "\nGoal reached.\n";

    } else {

        cout << "No path found from " << start << " to " << goal << ".\n";

    }

    return 0;

}

