

| **Title:** Implementation of informed search algorithm(Greedy Best First search/A\*) |
| --- |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Objective:** Examine the efficiency and performance of informed search algorithms in solving various problems **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

| **Course Outcome** | **After successful completion of the course students should be able to** |
| --- | --- |
| **CO2** | Analyse and solve problems for goal based agent architecture (searching and planning algorithms) |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **“Artificial Intelligence: a Modern Approach” by Russell and Norving, Pearson education Publications**
2. **“Artificial Intelligence” By Rich and knight, Tata Mcgraw Hill Publications**
3. [**http://people.cs.pitt.edu/~milos/courses/cs2710/lectures/Class4.pdf**](http://people.cs.pitt.edu/~milos/courses/cs2710/lectures/Class4.pdf)
4. [**http://cs.williams.edu/~andrea/cs108/Lectures/InfSearch/infSearch.html**](http://cs.williams.edu/~andrea/cs108/Lectures/InfSearch/infSearch.html)
5. **http://www.cs.mcgill.ca/~dprecup/courses/AI/Lectures/ai-lecture02.pdf** [**http://homepage.cs.uiowa.edu/~hzhang/c145/notes/04a-search.pdf**](http://homepage.cs.uiowa.edu/~hzhang/c145/notes/04a-search.pdf)
6. [**http://wiki.answers.com/Q/Informed\_search\_techniques\_and\_uninformed\_search\_techniques**](http://wiki.answers.com/Q/Informed_search_techniques_and_uninformed_search_techniques)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Problem solving, state-space trees, problem formulation, goal based agent architecture

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Historical Profile:**

### Problem-Solving Agent

A problem-solving agent is designed to find solutions to well-defined problems. This agent typically follows these steps:

1. **Formulate the Problem**: Define the initial state, goal state, and possible actions.
2. **Search for a Solution**: Use an appropriate search strategy to explore the problem space.
3. **Execute the Solution**: Apply the sequence of actions derived from the search.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**New Concepts to be learned:**

Informed search.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Informed Searching Technique**

**Greedy Best-First Search:**

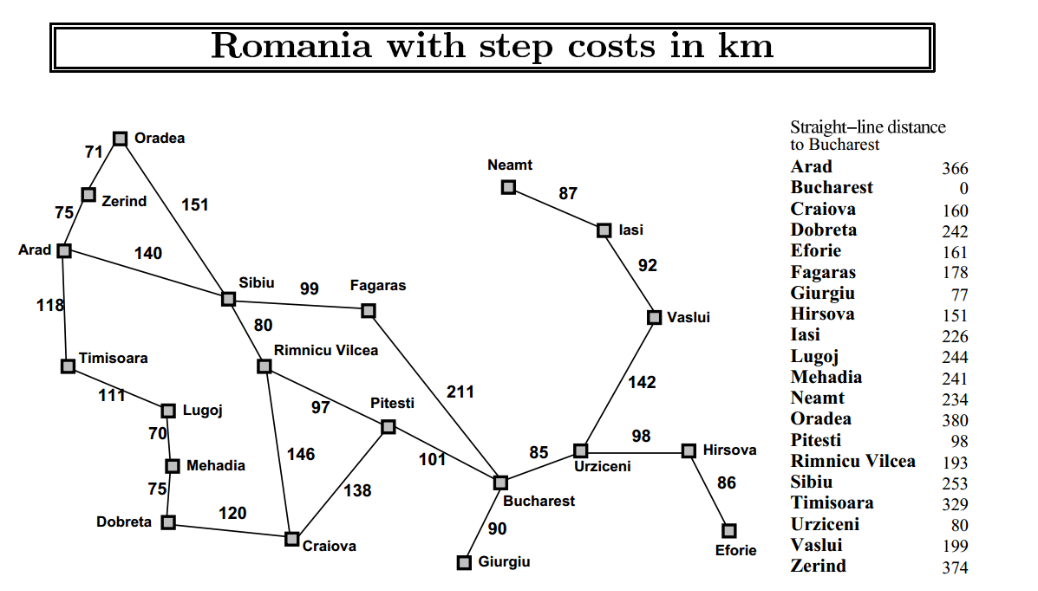
* Chooses the path that appears closest to the goal using a heuristic.
* Quick but might not find the optimal path.

***A*\* Search:**

* Combines path cost and heuristic to find the best path.
* Ensures the optimal path if the heuristic is accurate.

**Problem Statement:**

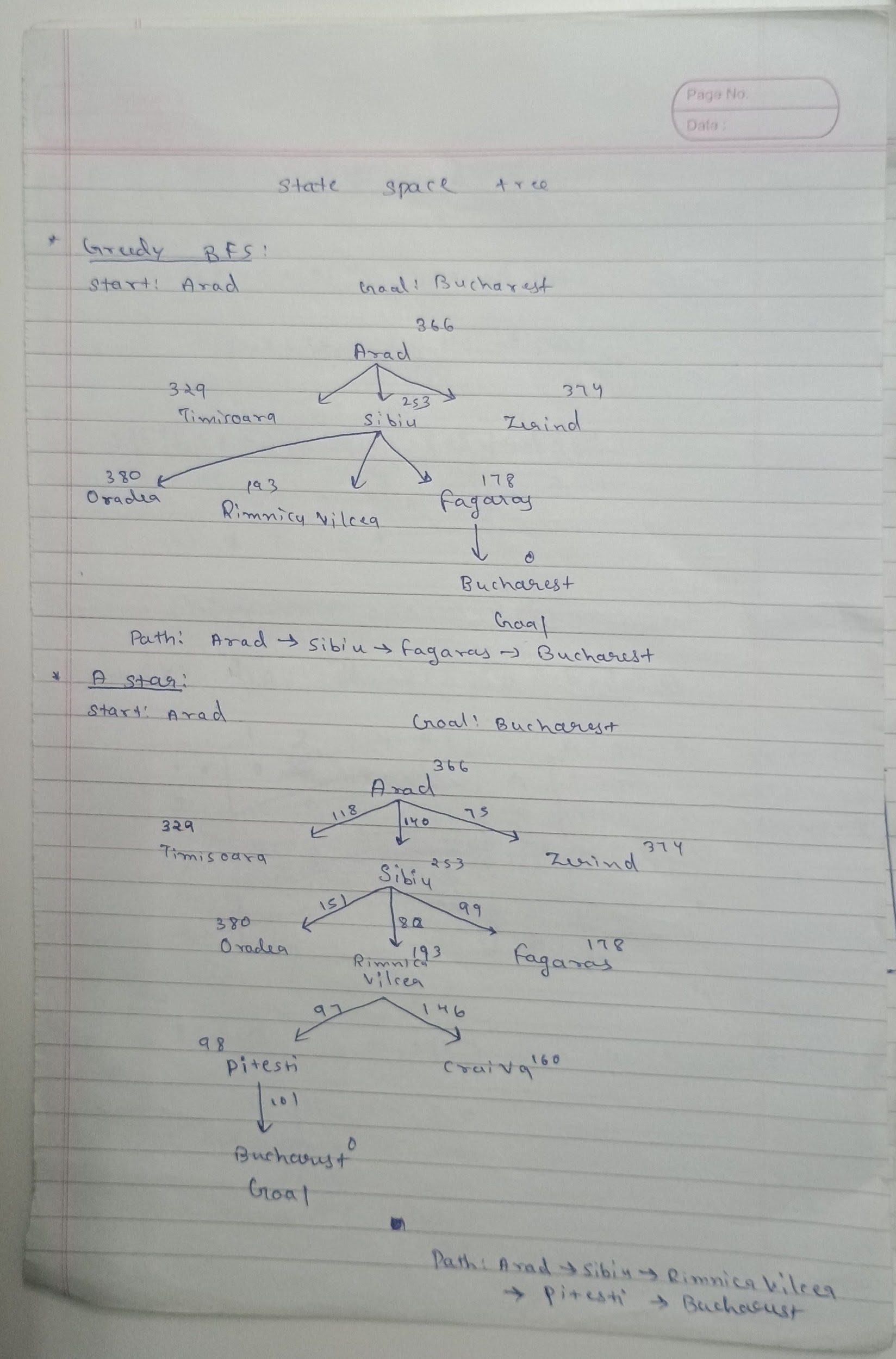
You are given a map of Romania with cities and the roads connecting them. Your task is to perform a Depth-First Search (DFS) starting from a given city and visit all the reachable cities in Romania. The goal is to simulate the DFS algorithm and return the order in which the cities are visited.

****

**Constraints:**

* The graph is bidirectional, meaning all cities are reachable from one another.
* **Greedy BFS** & **A\*** are to be implemented.
* The input will always contain a valid start city.

**State-space tree:**

****

**Solution with of chosen algorithm on the state-space tree:**

**CODE:**

**A] Greedy Best-First Search (GBFS):**

# include <bits/stdc++.h>

using namespace std;

unordered\_map<string, vector<pair<string, int>>> graph;

unordered\_map<string, int> heuristic;

unordered\_map<string, string> parent;

unordered\_map<string, int> costSoFar;

void addEdge(const string &city1, const string &city2, int cost)

{

graph[city1].push\_back({city2, cost});

graph[city2].push\_back({city1, cost});

}

void printPath(const string &start, const string &goal, int totalCost)

{

vector<string> path;

string current = goal;

while (current != start)

{

path.push\_back(current);

current = parent[current];

}

path.push\_back(start);

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

for (const string &city : path)

{

cout << city << " ";

}

cout << "\nTotal cost: " << totalCost << endl;

}

struct Compare

{

bool operator()(const pair<string, int> &a, const pair<string, int> &b)

{

return a.second > b.second;

}

};

void greedyBFS(const string &start, const string &goal)

{

priority\_queue<pair<string, int>, vector<pair<string, int>>, Compare> pq;

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

costSoFar[start] = 0;

while (!pq.empty())

{

string current = pq.top().first;

cout << "Visiting: " << current << " with heuristic: " << heuristic[current] << endl;

pq.pop();

if (current == goal)

{

cout << "\nPath found: ";

printPath(start, goal, costSoFar[goal]);

return;

}

int minCost = INT\_MAX;

string nextNode;

for (auto &neighbor : graph[current])

{

string nextCity = neighbor.first;

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

if (parent.find(nextCity) == parent.end())

{

costSoFar[nextCity] = newCost;

pq.push({nextCity, heuristic[nextCity]});

parent[nextCity] = current;

if (heuristic[nextCity] < minCost)

{

minCost = heuristic[nextCity];

nextNode = nextCity;

}

}

}

if (!nextNode.empty())

cout << "Minimum heuristic cost node chosen: " << nextNode << " with cost: " << minCost << endl;

cout << endl;

}

cout << "\nGoal not reachable from start." << endl;

}

int main()

{

addEdge("Arad", "Zerind", 75);

addEdge("Arad", "Timisoara", 118);

addEdge("Arad", "Sibiu", 140);

addEdge("Zerind", "Oradea", 71);

addEdge("Oradea", "Sibiu", 151);

addEdge("Sibiu", "Fagaras", 99);

addEdge("Sibiu", "Rimnicu Vilcea", 80);

addEdge("Rimnicu Vilcea", "Pitesti", 97);

addEdge("Rimnicu Vilcea", "Craiova", 146);

addEdge("Craiova", "Drobeta", 120);

addEdge("Drobeta", "Mehadia", 75);

addEdge("Mehadia", "Lugoj", 70);

addEdge("Lugoj", "Timisoara", 111);

addEdge("Fagaras", "Bucharest", 211);

addEdge("Pitesti", "Bucharest", 101);

addEdge("Bucharest", "Giurgiu", 90);

addEdge("Bucharest", "Urziceni", 85);

addEdge("Urziceni", "Vaslui", 142);

addEdge("Urziceni", "Hirsova", 98);

addEdge("Hirsova", "Eforie", 86);

addEdge("Vaslui", "Lasi", 92);

addEdge("Lasi", "Neamt", 87);

heuristic["Arad"] = 366;

heuristic["Bucharest"] = 0;

heuristic["Craiova"] = 160;

heuristic["Drobeta"] = 242;

heuristic["Eforie"] = 161;

heuristic["Fagaras"] = 178;

heuristic["Giurgiu"] = 77;

heuristic["Hirsova"] = 151;

heuristic["Lasi"] = 226;

heuristic["Lugoj"] = 244;

heuristic["Mehadia"] = 241;

heuristic["Neamt"] = 234;

heuristic["Oradea"] = 380;

heuristic["Pitesti"] = 98;

heuristic["Rimnicu Vilcea"] = 193;

heuristic["Sibiu"] = 253;

heuristic["Timisoara"] = 329;

heuristic["Urziceni"] = 80;

heuristic["Vaslui"] = 199;

heuristic["Zerind"] = 374;

cout << "Edges in the Map of Romania: " << endl << endl;

for (auto &city : graph)

{

cout << city.first << " -> ";

for (auto &neighbour : city.second)

{

cout << "(" << neighbour.first << ", " << neighbour.second << ") ";

}

cout << endl;

}

cout << endl;

string start, goal;

cout << "Enter the start city: ";

cin >> start;

cout << "Enter the goal city: ";

cin >> goal;

cout << endl;

if (graph.find(start) == graph.end() || graph.find(goal) == graph.end())

{

cout << "Start or Goal state not found!" << endl;

return 0;

}

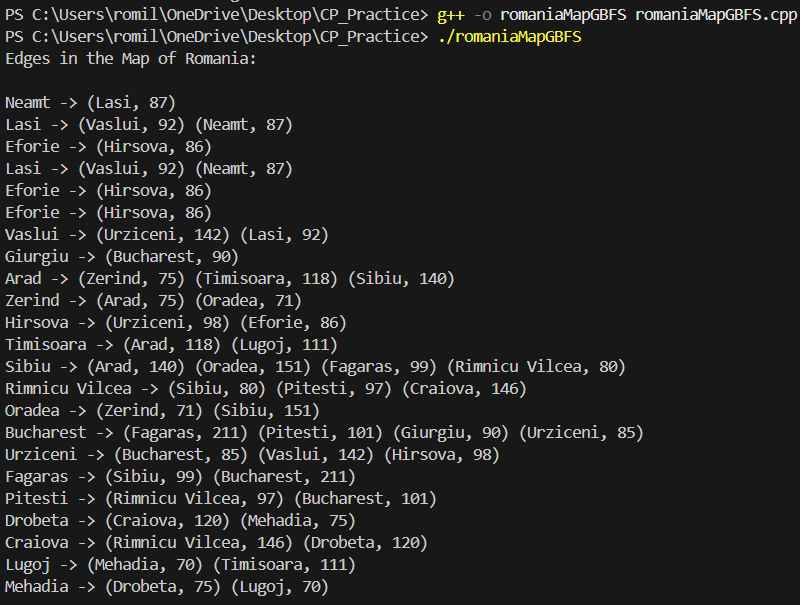
cout << endl;

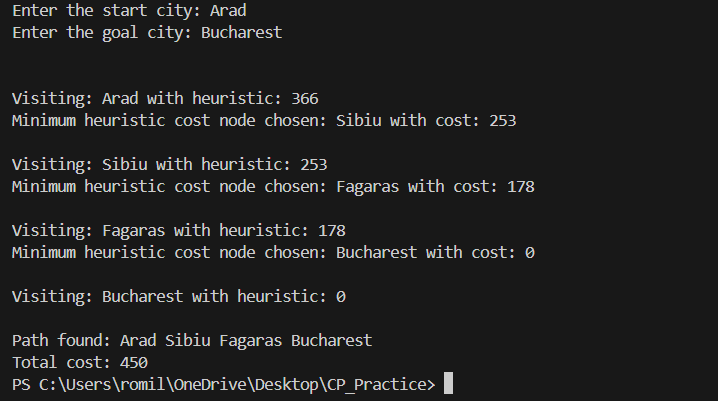
greedyBFS(start, goal);

return 0;

}

**OUTPUT:**

****

****

**B] A\* Algorithm:**

#include <bits/stdc++.h>

using namespace std;

unordered\_map<string, vector<pair<string, int>>> graph;

unordered\_map<string, int> heuristic;

unordered\_map<string, int> gScore;

unordered\_map<string, string> parent;

void addEdge(const string &city1, const string &city2, int cost)

{

graph[city1].push\_back({city2, cost});

graph[city2].push\_back({city1, cost});

}

void printPath(const string &start, const string &goal)

{

vector<string> path;

string current = goal;

while (current != start)

{

path.push\_back(current);

current = parent[current];

}

path.push\_back(start);

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

for (const string &city : path)

{

cout << city << " ";

}

cout << endl;

}

struct Compare

{

bool operator()(const pair<string, int> &a, const pair<string, int> &b)

{

return a.second > b.second;

}

};

void aStar(const string &start, const string &goal)

{

priority\_queue<pair<string, int>, vector<pair<string, int>>, Compare> pq;

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

gScore[start] = 0;

while (!pq.empty())

{

string current = pq.top().first;

cout << "Visiting: " << current << " with heuristic: " << heuristic[current] << endl;

pq.pop();

if (current == goal)

{

cout << endl << "Path found: ";

printPath(start, goal);

return;

}

for (auto &neighbor : graph[current])

{

string nextCity = neighbor.first;

int cost = neighbor.second;

int tentative\_gScore = gScore[current] + cost;

if (gScore.find(nextCity) == gScore.end() || tentative\_gScore < gScore[nextCity])

{

gScore[nextCity] = tentative\_gScore;

int fScore = tentative\_gScore + heuristic[nextCity];

cout << "Considering: " << nextCity << " with heuristic: " << heuristic[nextCity] << " and fScore: " << fScore << endl;

pq.push({nextCity, fScore});

parent[nextCity] = current;

}

}

cout << endl;

}

cout << endl << "Goal not reachable from start." << endl;

}

int main()

{

addEdge("Arad", "Zerind", 75);

addEdge("Arad", "Timisoara", 118);

addEdge("Arad", "Sibiu", 140);

addEdge("Zerind", "Oradea", 71);

addEdge("Oradea", "Sibiu", 151);

addEdge("Sibiu", "Fagaras", 99);

addEdge("Sibiu", "Rimnicu Vilcea", 80);

addEdge("Rimnicu Vilcea", "Pitesti", 97);

addEdge("Rimnicu Vilcea", "Craiova", 146);

addEdge("Craiova", "Drobeta", 120);

addEdge("Drobeta", "Mehadia", 75);

addEdge("Mehadia", "Lugoj", 70);

addEdge("Lugoj", "Timisoara", 111);

addEdge("Fagaras", "Bucharest", 211);

addEdge("Pitesti", "Bucharest", 101);

addEdge("Bucharest", "Giurgiu", 90);

addEdge("Bucharest", "Urziceni", 85);

addEdge("Urziceni", "Vaslui", 142);

addEdge("Urziceni", "Hirsova", 98);

addEdge("Hirsova", "Eforie", 86);

addEdge("Vaslui", "Lasi", 92);

addEdge("Lasi", "Neamt", 87);

heuristic["Arad"] = 366;

heuristic["Bucharest"] = 0;

heuristic["Craiova"] = 160;

heuristic["Drobeta"] = 242;

heuristic["Eforie"] = 161;

heuristic["Fagaras"] = 178;

heuristic["Giurgiu"] = 77;

heuristic["Hirsova"] = 151;

heuristic["Lasi"] = 226;

heuristic["Lugoj"] = 244;

heuristic["Mehadia"] = 241;

heuristic["Neamt"] = 234;

heuristic["Oradea"] = 380;

heuristic["Pitesti"] = 98;

heuristic["Rimnicu Vilcea"] = 193;

heuristic["Sibiu"] = 253;

heuristic["Timisoara"] = 329;

heuristic["Urziceni"] = 80;

heuristic["Vaslui"] = 199;

heuristic["Zerind"] = 374;

string start, goal;

cout << "Enter the start city: ";

cin >> start;

cout << "Enter the goal city: ";

cin >> goal;

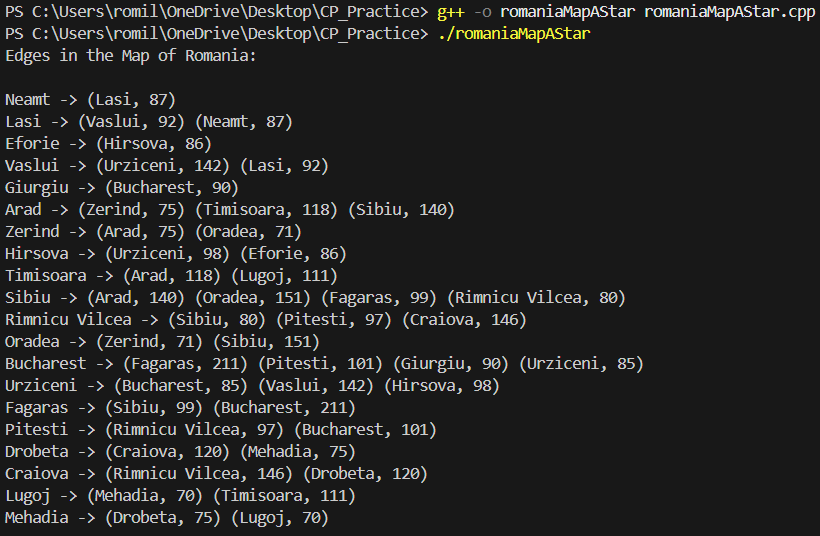
cout << endl;

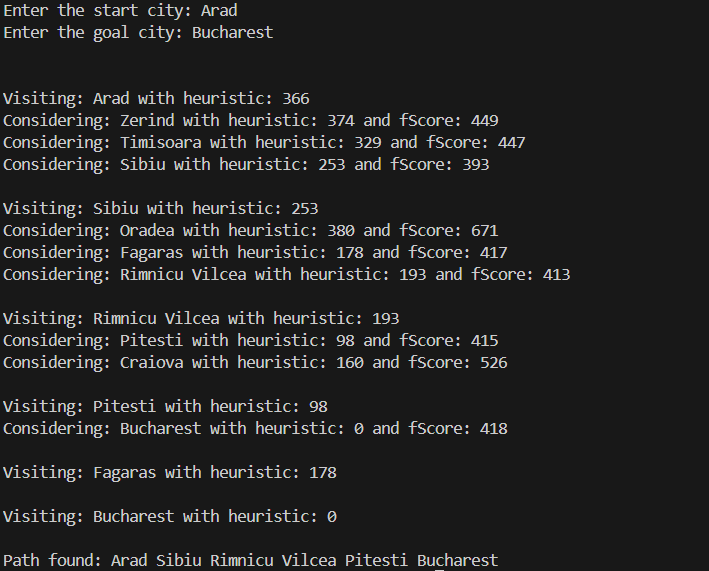
aStar(start, goal);

return 0;

}

**OUTPUT:**





**Comparison of Performance:** Greedy Best-First Search vs. A\* Algorithm:

| **Property** | **Greedy Best-First Search (GBFS)** | **A\* Algorithm** |
| --- | --- | --- |
| **Search Strategy** | Expands the node that appears closest to the goal (based only on heuristic). | Expands nodes based on both cost from the start (**gScore**) and estimated cost to goal (**heuristic**). |
| **Optimality** | Not guaranteed to find the shortest path; may get stuck in local optima. | Always finds the shortest path if the heuristic is admissible and consistent. |
| **Efficiency** | Faster than **A\*** in many cases but may explore unnecessary nodes. | More efficient than **GBFS** because it balances cost and heuristic. |
| **Memory Usage** | Requires less memory since it stores fewer path costs. | Uses more memory to store **gScore**, **fScore**, and parent tracking. |
| **Completeness** | May fail if it gets stuck in loops or local optima. | Always complete if a solution exists. |
| **Performance in Large Graphs** | Can be very fast but may take inefficient paths. | More computationally expensive but ensures the optimal path. |

**When to Use Each Algorithm:**

* Use **Greedy BFS** when speed is more critical than accuracy and you don’t need the absolute shortest path.
* Use **A\*** when you need the optimal path and can afford extra computation.

**Properties of A\* Algorithm:**

1. **Admissibility**
   * **A\*** is admissible if the heuristic function h(n) never overestimates the actual cost to reach the goal.
   * Ensures the algorithm finds the shortest path.
2. **Consistency (Monotonicity)**
   * A heuristic is consistent if, for every node n and its successor m, the estimated cost from n to the goal is never greater than the step cost plus the estimated cost from m to the goal.
   * If a heuristic is consistent, **A\*** does not need to revisit nodes.
3. **Completeness**
   * **A\*** is complete, meaning it always finds a solution if one exists.
4. **Optimality**
   * Guarantees the shortest path if the heuristic is admissible and consistent.
5. **Time Complexity**
   * Depends on the heuristic. If h(n) is too weak, **A\*** behaves like Dijkstra’s algorithm (expensive). If h(n) is too strong, it may behave like **GBFS**.
6. **Space Complexity**
   * Stores all explored nodes in memory, leading to high space complexity (**O(bd)**, where b is branching factor and d is depth).
7. **Trade-off Between Speed and Accuracy**
   * Balances between exploration (**gScore**) and goal direction (**heuristic**), making it more efficient than brute-force algorithms like Dijkstra’s.

**Post lab Objective questions**

1. **A heuristic is a way of trying**
2. To discover something or an idea embedded in a program
3. To search and measure how far a node in a search tree seems to be from a goal
4. To compare two nodes in a search tree to see if one is better than the other
5. Only (a) and (b)
6. Only (a), (b) and (c).
7. **A\* algorithm is based on**
8. Breadth-First-Search
9. Depth-First –Search
10. Best-First-Search
11. Hill climbing.
12. Bulkworld Problem.

**3. What is a heuristic function?**

1. A function to solve mathematical problems
2. A function which takes parameters of type string and returns an integer value
3. A function whose return type is nothing
4. A function which returns an object
5. A function that maps from problem state descriptions to measures of desirability.

**Post Lab Subjective Questions:**

1. **How does the Greedy Best-First Search algorithm use a heuristic evaluation function?**

* Greedy Best-First Search (GBFS) selects the node that appears closest to the goal based on the heuristic function h(n).
* It does not consider the cost from the start state, which can lead to suboptimal paths.
* It focuses only on minimizing h(n), which may result in getting stuck in local optima or inefficient paths.

1. **Find a good heuristic function for following**

* **Monkey and Banana Problem**
  + The heuristic can be the Manhattan distance between the monkey and the banana.
  + The fewer actions required (climbing, moving, grabbing) to reach the goal, the lower the heuristic value.
* **Travelling Salesman Problem (TSP)**
  + A good heuristic is the Minimum Spanning Tree (MST) heuristic, which estimates the cost by computing the MST of the remaining cities.
  + Another heuristic is the sum of the nearest neighbor distances, giving an approximate lower bound for the shortest tour.

1. **Define the heuristic search. Discuss benefits and shortcomings.**

**Definition:**  
Heuristic search is a technique that uses **domain-specific knowledge** (heuristic function) to guide the search process toward the goal more efficiently than uninformed search strategies.

**Benefits:**

* Reduces the **search space**, improving efficiency.
* Finds solutions **faster** than brute-force methods.
* Helps solve **complex problems** where exhaustive search is impractical.

**Shortcomings:**

* May not always find the **optimal** solution (depends on heuristic quality).
* Some heuristics can lead to **misleading paths** or local optima.
* **Computational overhead** if the heuristic function is complex.