K. J. Somaiya College of Engineering, Mumbai-77

Batch:	A-4	**Roll No.:**	16010122151

Experiment / assignment / tutorial No. 5

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

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**
4. **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
6. **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.

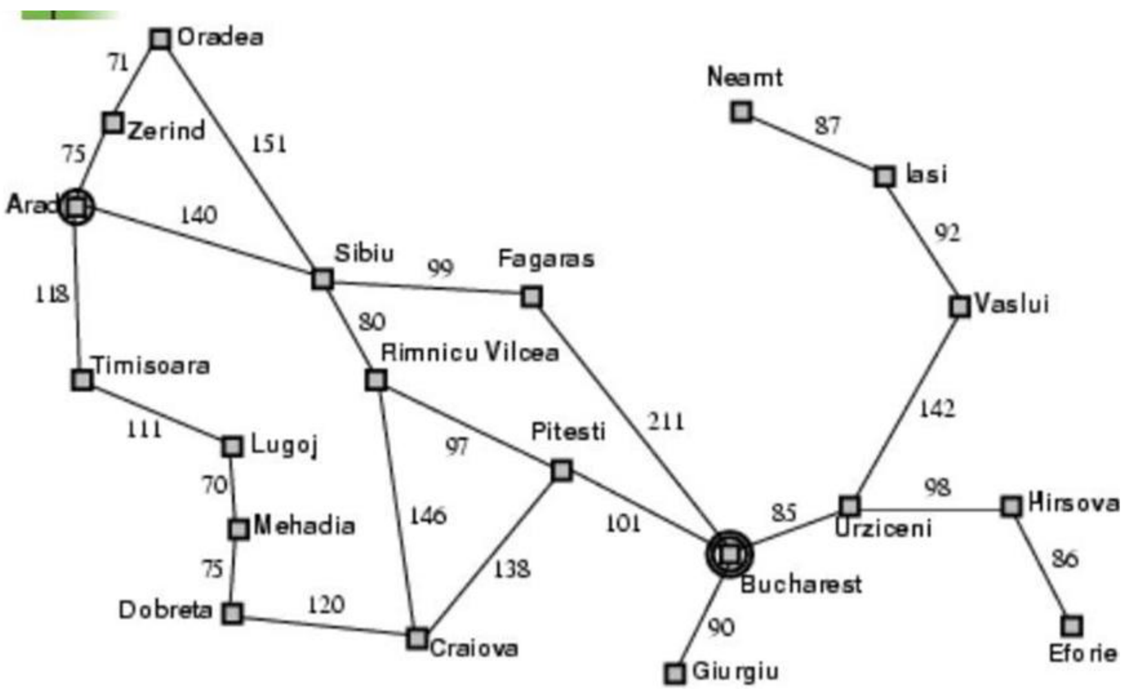
A* **Search:**

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

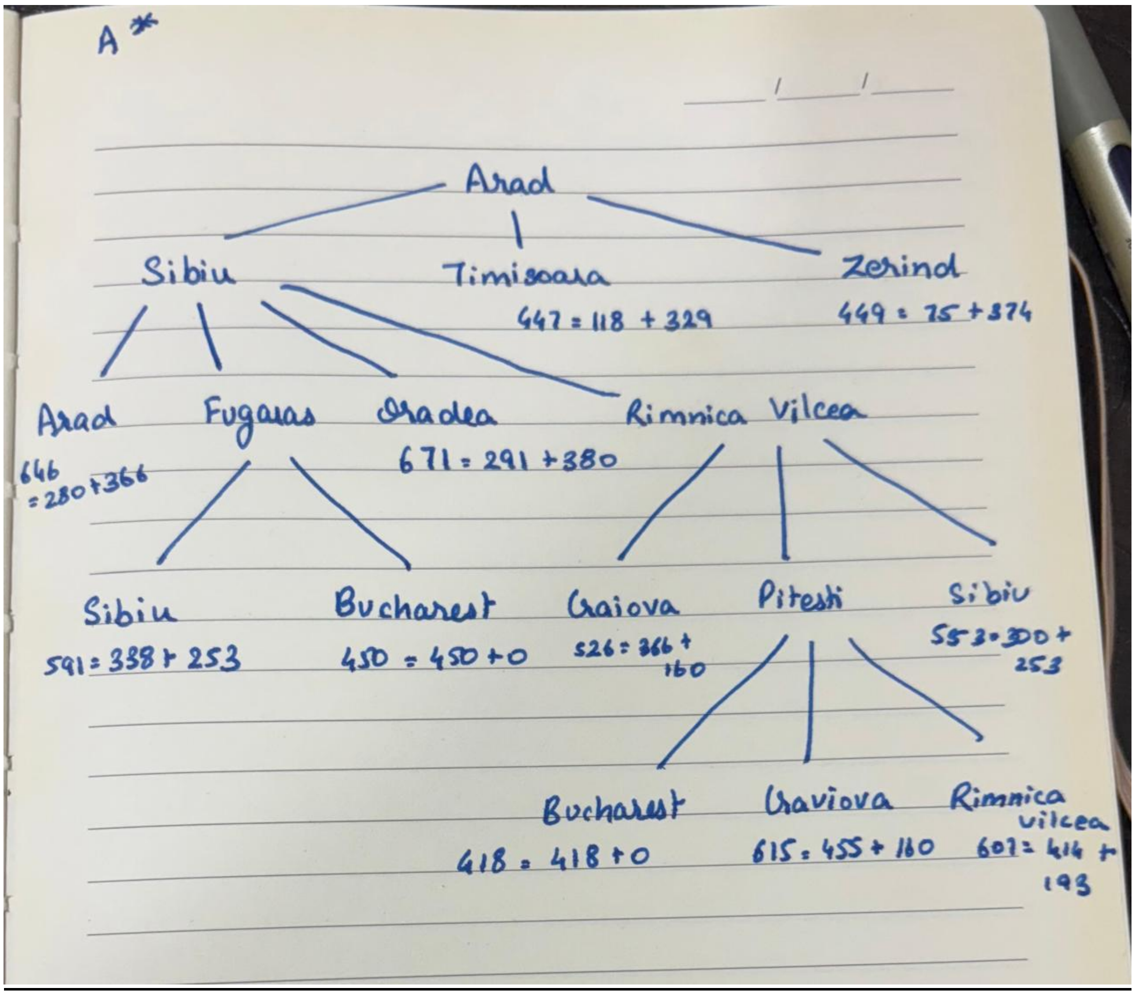
Problem Statement:

Implement the A* search algorithm to find the shortest path between two cities on the Map of Romania, using road distances as actual costs and straight-line distances to the goal city as heuristic estimates.

State-space tree :



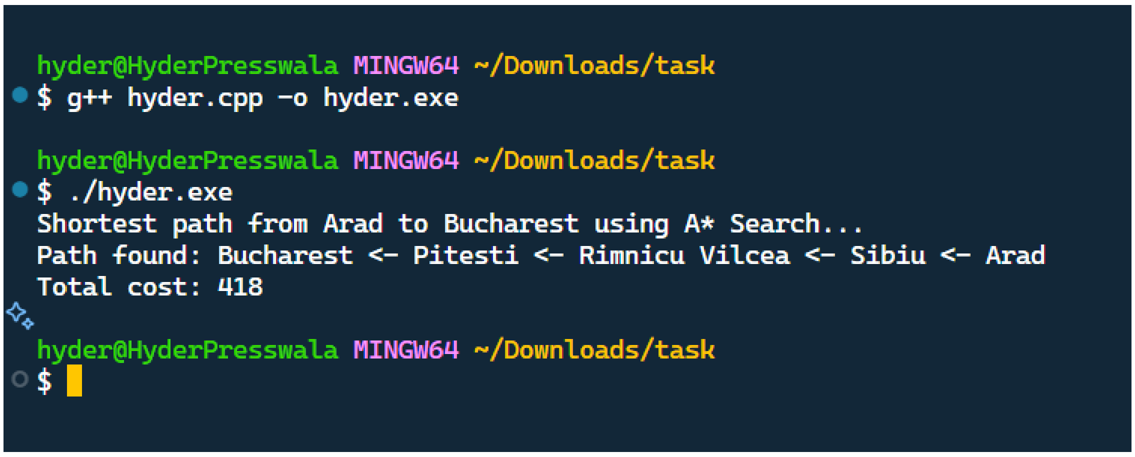
Solution with of chosen algorithm on the state-space tree: _(attach solution solved on paper)_



Source code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <limits>

using namespace std;

// Structure to represent a city with its name and cost
struct Node {
    string city;
    int costFromStart;      // Cost from start node
    int heuristicValue;     // Heuristic (straight-line distance to
goal)
    int cost;               // Total cost (costFromStart +
heuristicValue)

    bool operator>(const Node& other) const {
        return cost > other.cost; // Priority queue uses greater for
min-heap
    }
};

// Romania map graph (adjacency list with distances)
unordered_map<string, vector<pair<string, int>>> graph = {
    {"Arad", {{"Zerind", 75}, {"Sibiu", 140}, {"Timisoara", 118}}},
    {"Zerind", {{"Arad", 75}, {"Oradea", 71}}},
    {"Oradea", {{"Zerind", 71}, {"Sibiu", 151}}},
    {"Sibiu", {{"Arad", 140}, {"Oradea", 151}, {"Fagaras", 99},
{"Rimnicu Vilcea", 80}}},
    {"Timisoara", {{"Arad", 118}, {"Lugoj", 111}}},
    {"Lugoj", {{"Timisoara", 111}, {"Mehadia", 70}}},
    {"Mehadia", {{"Lugoj", 70}, {"Drobeta", 75}}},
    {"Drobeta", {{"Mehadia", 75}, {"Craiova", 120}}},
    {"Craiova", {{"Drobeta", 120}, {"Rimnicu Vilcea", 146},
{"Pitesti", 138}}},
    {"Rimnicu Vilcea", {{"Sibiu", 80}, {"Craiova", 146}, {"Pitesti",
97}}},
    {"Fagaras", {{"Sibiu", 99}, {"Bucharest", 211}}},
    {"Pitesti", {{"Rimnicu Vilcea", 97}, {"Craiova", 138},
{"Bucharest", 101}}},
    {"Bucharest", {{"Fagaras", 211}, {"Pitesti", 101}, {"Giurgiu",
90}, {"Urziceni", 85}}},
    {"Giurgiu", {{"Bucharest", 90}}},
    {"Urziceni", {{"Bucharest", 85}, {"Hirsova", 98}, {"Vaslui",
142}}},
    {"Hirsova", {{"Urziceni", 98}, {"Eforie", 86}}},
    {"Eforie", {{"Hirsova", 86}}},
    {"Vaslui", {{"Urziceni", 142}, {"Iasi", 92}}},
    {"Iasi", {{"Vaslui", 92}, {"Neamt", 87}}},
    {"Neamt", {{"Iasi", 87}}}
};

// Heuristic function (Straight-line distance to Bucharest)
unordered_map<string, int> heuristic = {
    {"Arad", 366}, {"Zerind", 374}, {"Oradea", 380}, {"Sibiu", 253},
    {"Timisoara", 329}, {"Lugoj", 244}, {"Mehadia", 241},
{"Drobeta", 242},
    {"Craiova", 160}, {"Rimnicu Vilcea", 193}, {"Fagaras", 178},
{"Pitesti", 98},
    {"Bucharest", 0}, {"Giurgiu", 77}, {"Urziceni", 80}, {"Hirsova",
151},
    {"Eforie", 161}, {"Vaslui", 199}, {"Iasi", 226}, {"Neamt", 234}
};

// A* Search Algorithm
void aStarSearch(string start, string goal) {
    priority_queue<Node, vector<Node>, greater<Node>> openList;
    unordered_map<string, int> gScore;
    unordered_map<string, string> cameFrom;

    gScore[start] = 0;
    openList.push({start, 0, heuristic[start], heuristic[start]});

    while (!openList.empty()) {
        Node current = openList.top();
        openList.pop();

        if (current.city == goal) {
            cout << "Path found: ";
            string path = goal;
            while (cameFrom.find(path) != cameFrom.end()) {
                cout << path << " <- ";
                path = cameFrom[path];
            }
            cout << start << endl;
            cout << "Total cost: " << gScore[goal] << endl;
            return;
        }

        for (auto neighbor : graph[current.city]) {
            string neighborCity = neighbor.first;
            int distance = neighbor.second;
            int tentative_costFromStart = gScore[current.city] + 
distance;

            if (gScore.find(neighborCity) == gScore.end() ||
tentative_costFromStart < gScore[neighborCity]) {
                gScore[neighborCity] = tentative_costFromStart;
                int totalCost = tentative_costFromStart + 
heuristic[neighborCity];
                openList.push({neighborCity,
tentative_costFromStart, heuristic[neighborCity], totalCost});
                cameFrom[neighborCity] = current.city;
            }
        }
    }
    cout << "No path found!" << endl;
}

int main() {
    string start = "Arad";
    string goal = "Bucharest";
    cout << "Shortest path from " << start << " to " << goal << " 
using A* Search...\n";
    aStarSearch(start, goal);
    return 0;
}
```

Output screenshots:



```
hyder@HyderPresswala MINGW64 ~/Downloads/task
$ g++ hyder.cpp -o hyder.exe

hyder@HyderPresswala MINGW64 ~/Downloads/task
$ ./hyder.exe
Shortest path from Arad to Bucharest using A* Search...
Path found: Bucharest <- Pitesti <- Rimnicu Vilcea <- Sibiu <- Arad
Total cost: 418

hyder@HyderPresswala MINGW64 ~/Downloads/task
$ 
```

Comparison of performance of Greedy and A* Algorithm:

Property	**Greedy Best-First Search (GBFS)**	***A Algorithm****
Heuristic Used	Only heuristic (h(n) $h(n)$)	f(n)=g(n)+h(n) $f(n)=g(n)+h(n)$ (cost + heuristic)
Optimality	Not optimal (may not find the best path)	Optimal (if heuristic is admissible)
Completeness	No (can get stuck in loops)	Yes (if heuristic is consistent and finite branching)
Time Complexity	O(bm) $O(b^m)$ (worst-case, like DFS)	O(bd) $O(b^d)$ (exponential, but better with good heuristic)
Space Complexity	O(bm) $O(b^m)$ (stores all nodes)	O(bd) $O(b^d)$ (keeps all nodes in memory)

Efficiency	Fast but may not find best solution	Slower but guarantees optimal solution

Properties of A* algorithm:

1. **Complete (if branching factor is finite and heuristic is consistent).**
2. **Optimal (if the heuristic is admissible, i.e., never overestimates the true cost).**
3. **Optimally Efficient (no other optimal algorithm expands fewer nodes with the same heuristic).**
4. **Time and Space Complexity: Exponential**
5. $\mathbf{O}(b^d)$
6. $O(b^d)$**, but depends on heuristic quality.**
7. **Balances Exploration and Exploitation: Uses both known path cost and heuristic estimate.**

Post lab Objective questions

1. **A heuristic is a way of trying**
    a. To discover something or an idea embedded in a program
    b. To search and measure how far a node in a search tree seems to be from a goal
    c. To compare two nodes in a search tree to see if one is better than the other
    d. Only (a) and (b)
    e. Only (a), (b) and (c).

Answer: Only (a), (b), and (c)

0. **A* algorithm is based on**
    a. Breadth-First-Search
    b. Depth-First –Search
    c. Best-First-Search
    d. Hill climbing.
    e. Bulkworld Problem.

Answer: Best-First-Search

3. What is a heuristic function?

a. A function to solve mathematical problems
b. A function which takes parameters of type string and returns an integer value
c. A function whose return type is nothing
d. A function which returns an object
e. A function that maps from problem state descriptions to measures of desirability.

Answer: 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?**

- GBFS relies entirely on the heuristic (h(n)) to make decisions.
- It expands the node that appears closest to the goal, ignoring the actual path cost (g(n)).
- Advantage: Very fast in some cases (e.g., when the heuristic is well-aligned with the goal).
- Disadvantage: Can lead to suboptimal solutions or infinite loops if the heuristic is misleading.

2. **Find a good heuristic function for following**
    a. **Monkey and Banana problem**

In this problem, a monkey must navigate to reach a banana, often requiring it to move a box to climb on. A good heuristic function could be:

- Distance-Based Heuristic: The straight-line (Euclidean) or grid-based (Manhattan) distance from the monkey’s current position to the banana. If the monkey needs to push a box first, the heuristic could estimate the steps required to move the box into position plus the steps to climb it.

- Action-Based Heuristic: A count of the minimum actions needed (e.g., move to box, push box, climb) to reach the banana.

This heuristic helps guide the monkey efficiently but may not always account for obstacles, requiring adjustments for optimality.

b. Travelling Salesman problem

For TSP, where the goal is to find the shortest route visiting all cities once, effective heuristics include:

- Nearest Neighbor (Greedy Approach): Always travel to the closest unvisited city next. While fast, it may not yield the shortest overall path.
- Minimum Spanning Tree (MST) Cost: Estimates the lower bound of the remaining path cost by constructing a tree connecting all cities without cycles. This ensures the heuristic is admissible (never overestimates).
- Lin-Kernighan Heuristic: An advanced local search method that refines routes by swapping edges to minimize distance.

These heuristics balance speed and accuracy, though optimality is only guaranteed with exhaustive methods like Branch and Bound.

1. Define the heuristic search. Discuss benefits and short comings.

Heuristic search refers to problem-solving techniques that use rule-of-thumb estimates (heuristics) to explore the most promising paths first, reducing search time compared to uninformed methods like BFS or DFS. Examples include A* (combines cost and heuristic) and Greedy Best-First Search (uses heuristic alone).

Benefits

- Speed: Dramatically reduces the search space by prioritizing likely solutions (e.g., A* in GPS navigation).
- Scalability: Handles large problem spaces (e.g., chess AI evaluating board states).
- Flexibility: Adaptable to diverse problems (pathfinding, puzzles, logistics) with tailored heuristics.

Shortcomings

- Suboptimal Solutions: Poor heuristics (e.g., Greedy BFS) may miss the best path.
- Design Complexity: Crafting effective heuristics requires domain expertise (e.g., admissible heuristics for A*).
- Computational Costs: Some heuristics (e.g., MST in TSP) are expensive to compute mid-search.