Bharatiya Vidya Bhavan's
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

SUB: AIML
Branch: COMPS B

Name :	**Chetan Deepak Patil**
UID :	**2023301012**
Exp no :	**03**
Aim :	**Implement a the Road Map problem using the Informed searching technique using A* search. The start node is Arad and Goal Node is Bucharest. Analyze the algorithm with respect to Completeness, Optimality, time and space Complexity.**

Thesis :

Explanation of the Code and Experiment

Objective: The objective of this experiment is to solve the "Road Map Problem" using the A* search algorithm, an informed search technique, to find the shortest path from Arad to Bucharest in the provided road map. A* search is widely used in pathfinding and graph traversal due to its efficiency and optimality in many scenarios.

Code Overview:

1. **Graph Representation:** The road map is represented as a graph where each city is a node (`Node` struct), and each road between two cities is an edge with an associated cost (distance). The graph also includes heuristic values for each node, which represent the estimated cost from that city to the goal city, Bucharest.
2. *A Search Algorithm:** The A* algorithm uses a combination of the actual cost (`gScore`) from the start node to a current node and an estimated cost (`heuristic`) from the current node to the goal node to determine the shortest path. The total estimated cost, `fScore`, is used to prioritize nodes in a priority queue.
3. **Implementation Details:**
    - The algorithm begins by initializing the start node (Arad) and pushing it into the priority queue with its heuristic value.
    - A loop iterates over nodes in the priority queue, expanding the current node by exploring its neighbors.
    - For each neighboring node, the algorithm calculates the tentative

cost (`tentative_gScore`) from the start node through the current node. If this cost is lower than a previously recorded cost for that node, it updates the path and re-prioritizes the node in the queue.
    - The search terminates when the goal node (Bucharest) is reached or when there are no more nodes to explore.
4. **Output:** The output displays the shortest path from Arad to Bucharest along with the distance covered between each pair of cities and the total distance at the end.

Experiment Conclusion

Completeness: The A* search algorithm is complete, meaning it will always find a solution if one exists. In this experiment, it successfully finds the shortest path from Arad to Bucharest.

Optimality: The A* algorithm is optimal as it always finds the least-cost path to the goal if the heuristic used is admissible (never overestimates the actual cost). Here, the heuristic values represent straight-line distances to Bucharest, ensuring that the algorithm finds the shortest route.

Time Complexity: The time complexity of A* search depends on the branching factor (`b`), the depth of the optimal solution (`d`), and the quality of the heuristic. In the worst case, the time complexity is `O(b^d)`. However, with a good heuristic, the performance is significantly improved compared to uninformed search methods like Breadth-First Search (BFS) or Depth-First Search (DFS).

Space Complexity: A* requires storage for all generated nodes in the worst-case scenario, making its space complexity also `O(b^d)`. However, due to the heuristic-driven approach, the number of nodes processed is generally lower than in uninformed searches.

Learnings from the Experiment

1. **Heuristic-Driven Search:** The experiment demonstrates the power of heuristic-driven search techniques like A*. By utilizing domain-specific knowledge (the straight-line distance heuristic), A* significantly reduces the search space compared to uninformed methods.
2. **Optimal Pathfinding:** The experiment shows that A* not only finds a path but guarantees that it is the shortest one due to the admissible heuristic used. This property makes A* a preferred choice for pathfinding in scenarios where optimality is required.
3. **Balance Between Time and Space:** A* exemplifies a trade-off between time and space complexity. While it is efficient in terms of the number of nodes explored (time complexity), it can be space-intensive as it needs

to maintain information about all generated nodes. This balance is crucial when choosing algorithms for pathfinding in large graphs.

4. **Applicability in Real-World Problems:** The implementation and experiment highlight the practical application of A* in real-world scenarios such as navigation systems, robotics, and games. The ability to adapt to different heuristics allows A* to be versatile and efficient across various domains.

Code :

```
#include <iostream>

#include <vector>

#include <queue>

#include <unordered_map>

#include <cmath>

#include <climits>

#include <algorithm>



using namespace std;



// Define a structure for representing the graph nodes

struct Node {

   string name;

   vector<pair<Node*, int>> neighbors; // Pair of (neighboring node, cost to that neighbor)

   int heuristic; // Heuristic value (straight-line distance to the goal)
```

```
    Node(string n, int h) : name(n), heuristic(h) {}
};

// Comparator for the priority queue to compare total costs
struct CompareCost {
    bool operator()(pair<Node*, int> a, pair<Node*, int> b) {
        return a.second > b.second; // Min-heap based on total cost
    }
};

// Function to perform the A* search
pair<vector<pair<string, int>>, int> aStarSearch(Node* start, Node*
goal) {
    // Priority queue to store (current node, total cost) pairs,
ordered by total cost
    priority_queue<pair<Node*, int>, vector<pair<Node*, int>>,
CompareCost> openSet;
    unordered_map<Node*, int> gScore; // Actual cost from start to
current node
    unordered_map<Node*, Node*> cameFrom; // To reconstruct the path
```

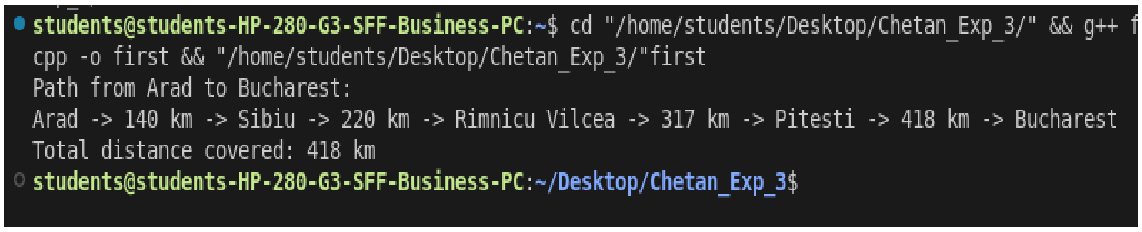
```
openSet.push({start, start->heuristic});

gScore[start] = 0;



while (!openSet.empty()) {

    Node* current = openSet.top().first;

    openSet.pop();



    if (current == goal) { // Goal reached

        vector<pair<string, int>> path;

        int totalDistance = gScore[goal];

        while (current != nullptr) {

            path.push_back({current->name, gScore[current]});

            current = cameFrom[current];

        }

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

        return {path, totalDistance};

    }



    for (auto& neighbor : current->neighbors) {

        Node* neighborNode = neighbor.first;

        int cost = neighbor.second;
```

```
            int tentative_gScore = gScore[current] + cost;


            if (gScore.find(neighborNode) == gScore.end() ||
tentative_gScore < gScore[neighborNode]) {
                cameFrom[neighborNode] = current;
                gScore[neighborNode] = tentative_gScore;
                int fScore = tentative_gScore +
neighborNode->heuristic;
                openSet.push({neighborNode, fScore});
            }
        }
    }
    return {{}, 0}; // Return empty path and zero distance if goal
not reachable
}


int main() {
    // Create nodes
    Node* Arad = new Node("Arad", 366);
    Node* Zerind = new Node("Zerind", 374);
    Node* Oradea = new Node("Oradea", 380);
    Node* Sibiu = new Node("Sibiu", 253);
```

```
   Node* Fagaras = new Node("Fagaras", 178);

   Node* RimnicuVilcea = new Node("Rimnicu Vilcea", 193);

   Node* Pitesti = new Node("Pitesti", 98);

   Node* Timisoara = new Node("Timisoara", 329);

   Node* Lugoj = new Node("Lugoj", 244);

   Node* Mehadia = new Node("Mehadia", 241);

   Node* Drobeta = new Node("Drobeta", 242);

   Node* Craiova = new Node("Craiova", 160);

   Node* Bucharest = new Node("Bucharest", 0);

   Node* Giurgiu = new Node("Giurgiu", 77);

   Node* Urziceni = new Node("Urziceni", 80);

   Node* Hirsova = new Node("Hirsova", 151);

   Node* Eforie = new Node("Eforie", 161);

   Node* Vaslui = new Node("Vaslui", 199);

   Node* Iasi = new Node("Iasi", 226);

   Node* Neamt = new Node("Neamt", 234);



   // Define edges (bidirectional)

   Arad->neighbors = {{Zerind, 75}, {Sibiu, 140}, {Timisoara,
118}};

   Zerind->neighbors = {{Arad, 75}, {Oradea, 71}};
   Oradea->neighbors = {{Zerind, 71}, {Sibiu, 151}};

   Sibiu->neighbors = {{Arad, 140}, {Oradea, 151}, {Fagaras, 99},
{RimnicuVilcea, 80}};

   Fagaras->neighbors = {{Sibiu, 99}, {Bucharest, 211}};

   RimnicuVilcea->neighbors = {{Sibiu, 80}, {Pitesti, 97},
{Craiova, 146}};

   Pitesti->neighbors = {{RimnicuVilcea, 97}, {Craiova, 138},
{Bucharest, 101}};

   Timisoara->neighbors = {{Arad, 118}, {Lugoj, 111}};

   Lugoj->neighbors = {{Timisoara, 111}, {Mehadia, 70}};

   Mehadia->neighbors = {{Lugoj, 70}, {Drobeta, 75}};

   Drobeta->neighbors = {{Mehadia, 75}, {Craiova, 120}};

   Craiova->neighbors = {{Drobeta, 120}, {RimnicuVilcea, 146},
{Pitesti, 138}};

   Bucharest->neighbors = {{Fagaras, 211}, {Pitesti, 101},
{Giurgiu, 90}, {Urziceni, 85}};

   Giurgiu->neighbors = {{Bucharest, 90}};

   Urziceni->neighbors = {{Bucharest, 85}, {Hirsova, 98}, {Vaslui,
142}};

   Hirsova->neighbors = {{Urziceni, 98}, {Eforie, 86}};

   Eforie->neighbors = {{Hirsova, 86}};

   Vaslui->neighbors = {{Urziceni, 142}, {Iasi, 92}};

   Iasi->neighbors = {{Vaslui, 92}, {Neamt, 87}};
```

```cpp
    Neamt->neighbors = {{Iasi, 87}};


    // Perform A* Search from Arad to Bucharest

    pair<vector<pair<string, int>>, int> result = aStarSearch(Arad,
Bucharest);

    vector<pair<string, int>> path = result.first;

    int totalDistance = result.second;


    if (!path.empty()) {

        cout << "Path from Arad to Bucharest: \n";

        for (size_t i = 0; i < path.size(); ++i) {

            cout << path[i].first;

            if (i < path.size() - 1) {

                cout << " -> " << path[i+1].second << " km -> ";

            }

        }

        cout << "\nTotal distance covered: " << totalDistance << "
km" << endl;

    } else {

        cout << "No path found from Arad to Bucharest." << endl;

    }
    // Free memory

    delete Arad; delete Zerind; delete Oradea; delete Sibiu; delete Fagaras;

    delete RimnicuVilcea; delete Pitesti; delete Timisoara; delete Lugoj;

    delete Mehadia; delete Drobeta; delete Craiova; delete Bucharest; delete Giurgiu;

    delete Urziceni; delete Hirsova; delete Eforie; delete Vaslui; delete Iasi; delete Neamt;



    return 0;

}
```

Output :



```
students@students-HP-280-G3-SFF-Business-PC:~$ cd "/home/students/Desktop/Chetan_Exp_3/" && g++ f
cpp -o first && "/home/students/Desktop/Chetan_Exp_3/"first
Path from Arad to Bucharest:
Arad -> 140 km -> Sibiu -> 220 km -> Rimnicu Vilcea -> 317 km -> Pitesti -> 418 km -> Bucharest
Total distance covered: 418 km
students@students-HP-280-G3-SFF-Business-PC:~/Desktop/Chetan_Exp_3$
```

Conclusion :

Conclusion

The A* the search algorithm provides an optimal and efficient

	solution for pathfinding problems, as demonstrated by the experiment on the road map from Arad to Bucharest. Its informed search technique balances exploration with efficiency by leveraging heuristics, making it an excellent choice for various real-world applications. Unlike Depth-First Search (DFS), which explores all possible paths, A* guarantees the shortest path while being significantly more efficient in terms of time and space, provided a suitable heuristic is available. This experiment enhances understanding of how intelligent search techniques can be applied to solve complex problems effectively.