**Assignment 4**

**Problem Statement:**  
To implement the **A\*** Algorithm for an application.

**Theory**

**1. Search Algorithms in AI**

* Search is a fundamental technique in Artificial Intelligence for problem solving.
* Classical search algorithms like BFS and DFS work but are inefficient in large state spaces.
* The **A\*** algorithm improves efficiency by using a heuristic to guide the search.

**2. A\* Algorithm**

* **A\*** is a best-first search algorithm that finds the shortest path from a start node to a goal node.
* It combines:
  + **g(n):** Cost to reach node n from start.
  + **h(n):** Heuristic estimate of cost to reach goal from n.
  + **f(n) = g(n) + h(n):** Estimated total cost.
* The heuristic function plays a crucial role. For grid-based problems, **Manhattan Distance** is commonly used:

**3. Applications of A\***

* Robot navigation
* Route planning in maps (e.g., Google Maps)
* Pathfinding in games

**Algorithm Steps**

1. Initialize the open list with the start node.
2. While the open list is not empty:
   * Select the node with the smallest f value.
   * If this node is the goal → reconstruct path and stop.
   * Otherwise, expand all valid neighbors and compute their f = g + h.
   * Add neighbors to open list if not already visited.
3. If goal not found and open list is empty → no solution exists.

**Code (C++ Implementation)**

#include <bits/stdc++.h>

using namespace std;

// Structure to store coordinates and path cost

struct Node {

int x, y;

int g, h, f;

Node\* parent;

};

// Compare function for priority queue

struct Compare {

bool operator()(Node\* a, Node\* b) {

return a->f > b->f;

}

};

// Manhattan distance heuristic

int heuristic(int x1, int y1, int x2, int y2) {

return abs(x1 - x2) + abs(y1 - y2);

}

// Check if cell is valid (within grid and not an obstacle)

bool isValid(int x, int y, int rows, int cols, vector<vector<int>>& grid) {

return (x >= 0 && y >= 0 && x < rows && y < cols && grid[x][y] == 0);

}

// A\* Search Algorithm

vector<pair<int,int>> aStarSearch(vector<vector<int>>& grid, pair<int,int> start, pair<int,int> goal) {

int rows = grid.size();

int cols = grid[0].size();

// Open list (priority queue)

priority\_queue<Node\*, vector<Node\*>, Compare> openList;

// Closed list

vector<vector<bool>> closed(rows, vector<bool>(cols, false));

// Start node

Node\* startNode = new Node{start.first, start.second, 0, 0, 0, nullptr};

startNode->h = heuristic(start.first, start.second, goal.first, goal.second);

startNode->f = startNode->g + startNode->h;

openList.push(startNode);

// Directions: up, down, left, right

int dx[4] = {-1, 1, 0, 0};

int dy[4] = {0, 0, -1, 1};

while (!openList.empty()) {

Node\* current = openList.top();

openList.pop();

int x = current->x;

int y = current->y;

if (x == goal.first && y == goal.second) {

// Reconstruct path

vector<pair<int,int>> path;

while (current != nullptr) {

path.push\_back({current->x, current->y});

current = current->parent;

}

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

return path;

}

closed[x][y] = true;

// Explore neighbors

for (int i = 0; i < 4; i++) {

int nx = x + dx[i], ny = y + dy[i];

if (isValid(nx, ny, rows, cols, grid) && !closed[nx][ny]) {

int gNew = current->g + 1;

int hNew = heuristic(nx, ny, goal.first, goal.second);

int fNew = gNew + hNew;

Node\* neighbor = new Node{nx, ny, gNew, hNew, fNew, current};

openList.push(neighbor);

}

}

}

return {}; // No path found

}

// ----------------------

// Example Application

// ----------------------

int main() {

// 0 = free cell, 1 = obstacle

vector<vector<int>> grid = {

{0, 0, 0, 0, 0},

{1, 1, 0, 1, 0},

{0, 0, 0, 0, 0},

{0, 1, 1, 1, 1},

{0, 0, 0, 0, 0}

};

pair<int,int> start = {0, 0}; // starting point

pair<int,int> goal = {4, 4}; // goal point

vector<pair<int,int>> path = aStarSearch(grid, start, goal);

if (!path.empty()) {

cout << "Shortest Path found:\n";

for (auto p : path) {

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

}

cout << endl;

} else {

cout << "No path found!" << endl;

}

return 0;

}

**Sample Output**

Shortest Path found:

(0,0) (0,1) (0,2) (1,2) (2,2) (2,1) (2,0) (3,0) (4,0) (4,1) (4,2) (4,3) (4,4)

**Conclusion**

* The **A\*** algorithm was successfully implemented to solve a **grid-based shortest path problem**.
* It uses **Manhattan distance** as a heuristic, guiding the search toward the goal efficiently.
* A\* guarantees the shortest path if the heuristic is **admissible** (never overestimates).
* Applications include **robot navigation, maps, and games**.