

Experiment No. 6

Aim

To write a program to implement the **A*** (A-star) algorithm for finding the shortest path between two nodes in a graph.

Theory

The **A*** algorithm is a **heuristic-based graph traversal** method used to find the **shortest path** efficiently.

It combines:

- **g(n)**: Cost from start node to current node
- **h(n)**: Estimated cost from current node to goal (heuristic)
- **f(n) = g(n) + h(n)**: Total estimated cost

The algorithm chooses the path with the **lowest f(n)** value at each step, ensuring both optimality and efficiency.

Properties:

- **Complete**: Finds a solution if one exists
 - **Optimal**: If heuristic is admissible (never overestimates)
 - **Time Complexity**: $O(b^d)$
 - **Space Complexity**: $O(b^d)$
-

Algorithm

1. Initialize **Open List** (start node) and **Closed List** (empty).
 2. While Open List is not empty:
 - Pick node with **lowest f(n)**.
 - If node is the **goal**, reconstruct and return path.
 - For each valid neighbor:
 - Compute **g**, **h**, and **f**.
 - Add or update neighbor in Open List.
 3. If Open List becomes empty → no path exists.
-

Code

C++ Program (Condensed Version)

```
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>
#include <set>
using namespace std;

struct Node {
    int x, y; float g, h, f; Node* parent;
    Node(int x, int y, Node* p=nullptr, float g=0, float h=0)
        : x(x), y(y), g(g), h(h), f(g+h), parent(p) {}
};

float heuristic(Node* a, Node* b) {
    return sqrt((a->x - b->x)*(a->x - b->x) + (a->y - b->y)*(a->y - b->y));
}

void printPath(Node* n) {
    if (!n) return; printPath(n->parent); cout << "(" << n->x << "," << n->y
    << ") ";
}

void aStar(vector<vector<int>>& grid, Node* start, Node* goal) {
    priority_queue<pair<float, Node*>, vector<pair<float, Node*>>,
greater<>> open;
    set<pair<int,int>> closed;
    open.push({start->f, start});

    int dir[4][2] = {{-1,0},{1,0},{0,-1},{0,1}};
    int r = grid.size(), c = grid[0].size();

    while (!open.empty()) {
        Node* cur = open.top().second; open.pop();
        if (cur->x == goal->x && cur->y == goal->y) { printPath(cur);
return; }
        closed.insert({cur->x, cur->y});

        for (auto& d : dir) {
            int nx = cur->x + d[0], ny = cur->y + d[1];
            if (nx<0 || ny<0 || nx>=r || ny>=c || grid[nx]
[ny]==1 || closed.count({nx,ny})) continue;
            float g = cur->g + 1, h = heuristic(new Node(nx,ny), goal);
            open.push({g+h, new Node(nx,ny,cur,g,h)});
        }
    }
}
```

```

        cout << "No path found!";
    }

    int main() {
        vector<vector<int>> grid = {
            {0,0,0,0,0},
            {1,1,0,1,0},
            {0,0,0,0,0},
            {0,1,1,1,0},
            {0,0,0,0,0}
        };
        Node *start = new Node(0,0), *goal = new Node(4,4);
        cout << "Path: "; aStar(grid, start, goal); cout << endl;
    }

```

Input/Output Example

Grid:

S	0	0	0	0
1	1	0	1	0
0	0	0	0	0
0	1	1	1	0
0	0	0	0	G

Output:

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

Analysis

- **Approach:** Uses heuristic-driven search with lowest $f = g + h$.
- **Time Complexity:** $(O(E \log V))$
- **Space Complexity:** $(O(V))$
- **Advantage:** Balances speed and accuracy efficiently.

Real-Life Applications

- **GPS & Navigation Systems**
 - **Game AI Pathfinding**
 - **Autonomous Robots & Drones**
-

Conclusion

The **A*** algorithm was successfully implemented to find the shortest path in a grid environment. The experiment demonstrates how combining actual and heuristic costs results in a fast and optimal pathfinding solution widely used in AI-based navigation systems.
