Alberto Escalante

Professor Dabish

CS 460

June 24, 2024

CS 460 Project Algorithm

Algorithm: A* algorithm

Welcome | A*.cpp

Users > albertoescalante > A*.cpp > a_star(const vector<vector<int>>&, pair<int,int>, pair<int,int>)

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>
#include <unordered_map>
#include <algorithm>

using namespace std;

// Node struct represents each cell in the grid
struct Node {
    int x, y;           // Coordinates of the node
    double g, h, f;     // g = cost from start to current node, h = heuristic cost to goal, f = g + h
    Node* parent;       // Pointer to the parent node for path reconstruction

    // Constructor for Node
    Node(int x, int y, double g = 0.0, double h = 0.0, Node* parent = nullptr)
        : x(x), y(y), g(g), h(h), f(g + h), parent(parent) {}

    // Operator overloading for priority queue to compare nodes based on f value
    bool operator>(const Node& other) const {
        return f > other.f;
    }
};

// Hash function for Node to use in unordered_map
struct NodeHash {
    size_t operator()(const Node* node) const {
        return hash<int>()(node->x) ^ hash<int>()(node->y);
    }
};

// Equality function for Node to use in unordered_map
struct NodeEqual {
    bool operator()(const Node* a, const Node* b) const {
        return a->x == b->x && a->y == b->y;
    }
};
```

```cpp

// Function to get the neighbors of a node that are walkable
vector<Node*> get_neighbors(Node* node, const vector<vector<int>>& grid) {
    vector<Node*> neighbors;
    int x = node->x;
    int y = node->y;
    int rows = grid.size();
    int cols = grid[0].size();

    // Define possible directions to move (up, down, left, right)
    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    // Check each direction to find valid neighbors
    for (auto& dir : directions) {
        int newX = x + dir.first;
        int newY = y + dir.second;

        // Check if the new position is within the grid and is walkable (grid value is 0)
        if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && grid[newX][newY] == 0) {
            neighbors.push_back(new Node(newX, newY));
        }
    }

    return neighbors;
}

// Heuristic function to estimate the cost from a node to the goal (using Manhattan distance)
double heuristic(Node* a, Node* b) {
    return abs(a->x - b->x) + abs(a->y - b->y);
}

// Function to reconstruct the path from the goal node to the start node
vector<pair<int, int>> reconstruct_path(Node* node) {
    vector<pair<int, int>> path;
    while (node) {
        path.emplace_back(node->x, node->y); // Add current node's coordinates to the path
        node = node->parent; // Move to the parent node
    }
    reverse(path.begin(), path.end()); // Reverse the path to get it from start to goal
    return path;
}
```

```cpp

// A* algorithm implementation
vector<pair<int, int>> a_star(const vector<vector<int>>& grid, pair<int, int> start, pair<int, int> goal) {
    // Initialize start and goal nodes
    Node* startNode = new Node(start.first, start.second);
    Node* goalNode = new Node(goal.first, goal.second);

    // Priority queue to store the open set of nodes (nodes to be explored)
    priority_queue<Node*, vector<Node*>, greater<Node*>> openSet;
    // Unordered map to keep track of all nodes created during the search
    unordered_map<Node*, Node*, NodeHash, NodeEqual> allNodes;

    openSet.push(startNode); // Add start node to the open set
    allNodes[startNode] = startNode; // Add start node to the map of all nodes

    // Main loop to process nodes in the open set
    while (!openSet.empty()) {
        Node* current = openSet.top(); // Get the node with the lowest f value
        openSet.pop(); // Remove the node from the open set

        // Check if we have reached the goal
        if (current->x == goalNode->x && current->y == goalNode->y) {
            auto path = reconstruct_path(current); // Reconstruct the path
            for (auto& pair : allNodes) {
                delete pair.first; // Clean up all nodes
            }
            return path; // Return the path
        }
```

```cpp

        // Get valid neighbors of the current node
        auto neighbors = get_neighbors(current, grid);
        for (auto& neighbor : neighbors) {
            double tentative_g = current->g + 1; // Cost to move to a neighbor

            // Check if this path to the neighbor is better than any previous one
            if (allNodes.find(neighbor) == allNodes.end() || tentative_g < neighbor->g) {
                neighbor->g = tentative_g;
                neighbor->h = heuristic(neighbor, goalNode);
                neighbor->f = neighbor->g + neighbor->h;
                neighbor->parent = current;

                openSet.push(neighbor); // Add neighbor to the open set
                allNodes[neighbor] = neighbor; // Add neighbor to the map of all nodes
            }
        }
    }

    // Cleanup in case no path is found
    for (auto& pair : allNodes) {
        delete pair.first; // Clean up all nodes
    }

    return {}; // No path found
}
```