Anders Ladow

CE 224 Final Project

Written Section

Introduction

The goal of this project was to implement the A* and Dijkstra's algorithm to find the shortest path between two points, represented by a grid. On this grid are obstacles represented by something, and it is the algorithm's goal to navigate the obstacles in the most efficient way possible. Originally, I wanted this grid and the resulting ideal path to be represented in a game engine, specifically Unreal engine. While I got a grid structure visually up and running, I struggled with implementing my graph class in a way Unreal could compile. Because of this, I decided to visually represent the algorithm's working directly within the C++ console.

Data Structure

The first step was to create a graph class that the algorithms could be implemented upon, specifically a graph class that is configured to a grid with (x,y) coordinates. The graph class is represented by a node containing an adjacency list, in which the list contains every node connected to it. Unless the node lies on the perimeter of the grid, it is connected to 4 other nodes, one above, one below, one right, and one left. The weight of the edges is either 0 or 1, depending on whether one of the nodes connected to it is an "obstacle".

Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm mainly works by checking through the adjacency lists of nodes, in which the path is initialized with a starting value and a goal value to reach. It uses a priority queue to store nodes that have been checked and prioritize certain nodes that should be checked next as it moves along the path. When checking through the adjacency list of one of the nodes, for each node connected, it calculates the total distance it would take to reach the neighbor node from the current node. It does this by adding the distance it took to get to the current node to the weight of the edge connecting the next node. If this new distance is less than what is currently known, then the next node is added to the priority queue, and the distance is updated to include this node. The algorithm repeats this process until the goal coordinates are reached.

A* Algorithm

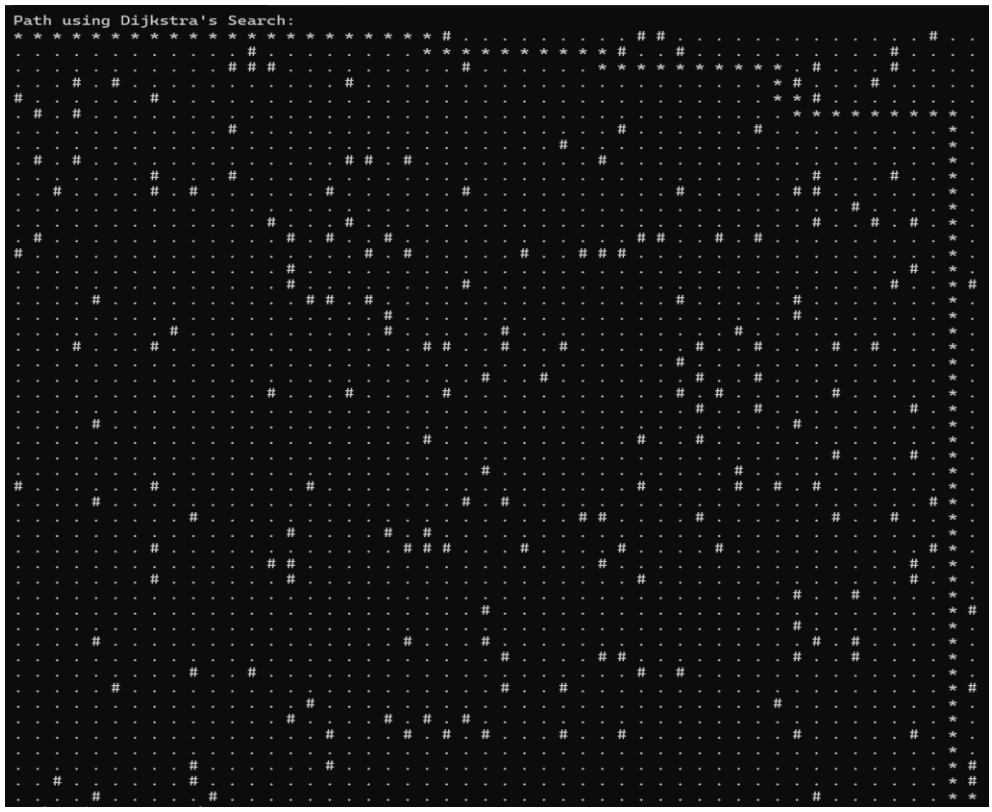
The A* algorithm works nearly identical to the Dijkstra's algorithm but has the addition of a heuristic function when prioritizing what node should be checked next. Many different heuristic functions can be used for the algorithm, and since this problem is represented by a 2D array, the heuristic function I chose to use was Euclidian distance, or straight-line distance between two coordinate points.

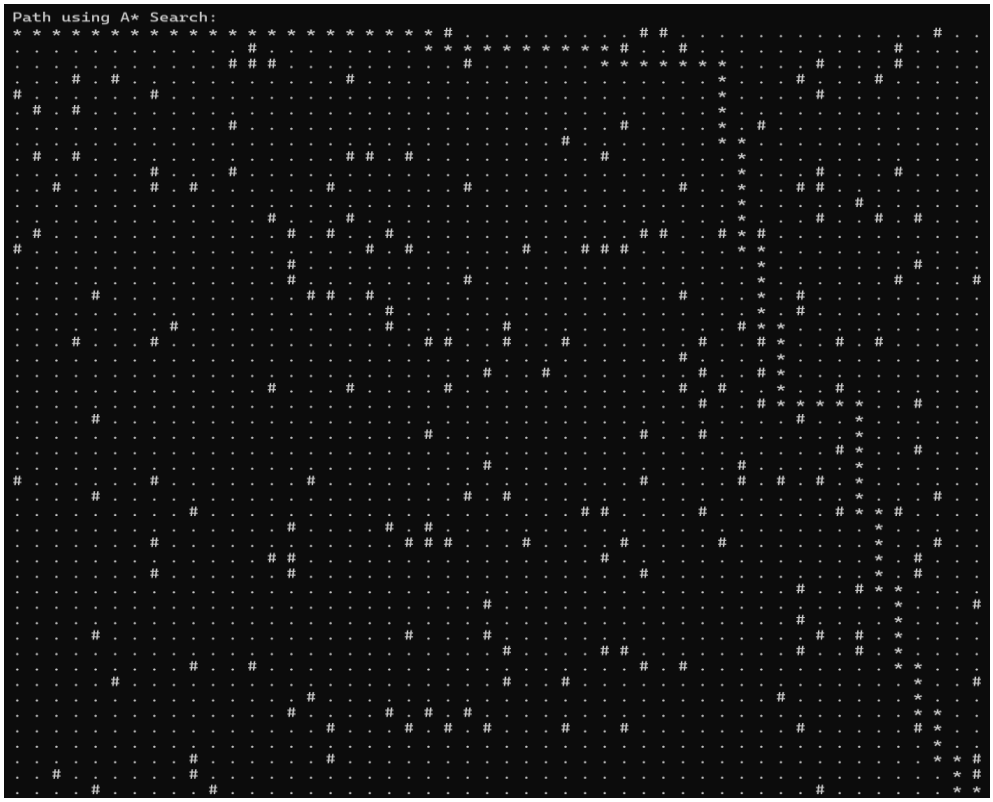
$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

When calculating the distance from the current node to a certain neighbor node, it now not only adds the weight of the edge connecting the nodes but now includes this heuristic distance in it's calculations.

Usage

As long as the two files, main.cpp and gridClass.cpp, are within the same directory, the code should run in something like visual studio. Since the obstacles are randomly generated, you can run the code multiple times to see how the paths change each time. You can also change the number of obstacles generated within the code in the main.cpp function to see how that affects the paths. (Note: If no path appears when running the code, it means that the randomly generated obstacles were placed in a way that closes off either the start or the goal node. I put in a little code to prevent this, but it is still possible. Restarting it should fix the problem). Running the code results in a path that resembles the following:


Path using Dijkstra's Search:


Path using A* Search: