CSSE451 - Motion Planning Project

Alec Tiefenthal

Jacob Knispel

Jayanth Shankar

Overview:

For this project, we were interested in studying motion planning, specifically with regards to graphical implementations. The main goal was to explore options in algorithms for quickly and tactically finding the best path through as many objectives as possible given a starting point and an environment with obstacles.

Ideally, the algorithm would also favor reaching a cluster of objectives over finding the optimal route through all of the objectives. In other words, the algorithm should act as though there is an unknown but limited amount of time to reach as many objectives as possible rather than acting as though it has all the time in the world to go out of its way to reach every objective.

Our project is made up of two major separate components: an implementation of A* search in a Javascript/HTML page, and a customized version of A* search using Unreal Engine (built on Unreal Engine's pathfinding algorithm).

Explanation of A*:

A* is a common algorithm used in pathfinding. Given a start node and an end node, it is guaranteed to find a path between the two (usually the optimal path) if such a path exists. Therefore, given a map made up of connected nodes with a start and end point(s), A* can be utilized to dynamically find a path from the start towards the end point to navigate around any obstacles.

The basic structure of A* is quite simple. For each node, starting at the starting node (the current position of our "robot"), looking to reach the end point, the algorithm examines all of our current point's neighbors, and evaluates their viability using a function f(x) , a combination of two factors:

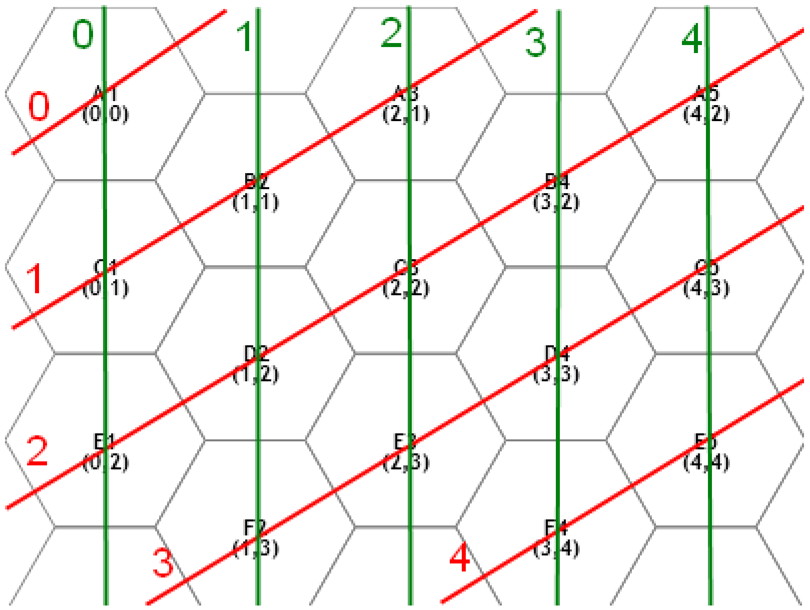
1. g(x): The cost to get from the start node to the current node. This is usually simply the number of "moves" it took to get from one to the other, but it could also take into account difficulty of terrain or similar attributes.
2. h(x): A heuristic used to estimate the cost from the current node to the destination node. This can be as simple as the direct distance between the two – the reason this isn't 100% accurate is because there may be obstacles in the way that obstruct a direct path.

The algorithm repeatedly picks the node with the lowest total sum of the two estimates (the lowest f(x)), and then repeats the process until there are no unvisited neighbors (in which case there is no solution) or reaches the end node.

A* Javascript Implementation:

Something great about the A* algorithm is that, being node-based, it does not require any coherent coordinate grid to work. In other words, the only important thing required to be able to run A* is a way to get all neighboring nodes of a given node. Even simpler: all you need is a graph of connected nodes. For this reason, it wasn't a great deal harder to implement A* in a hexagonal grid than it would be in normal Cartesian coordinate system.

The Javascript implementation of A* was written using hexagon grid code found here as a starting point, although it ended up being heavily modified. This was helpful because we didn't have to do very much of the strange hexagon math, and when we did we were able to use ideas from that project. The way a hexagon grid is treated easily as a coordinate plane is explained well by the picture to the right. More can be found at the link attached to the image, but essentially, the normal rows and columns found in a coordinate system are applied to the hexagon grid as shown so that each hexagon gets unique coordinates.



The first step to modifying this code to be capable of working for motion planning algorithms was to adapt the original author's code to be able to handle values being assigned to particular hexagons in the grid. The original code didn't represent hexagons with Javascript's object equivalent, meaning they couldn't be assigned values. Thus, the hex.js file was born, which holds functions that allow the hexagon to draw itself, find the distance to another hexagon, etc.

The next main step was to, well, implement the algorithm. Because we wanted it to demonstrate how it went about solving for the solution, this implementation was complicated a bit by calls to change cell color for display purposes. Now we'll go into detail about how this algorithm was actually implemented.

Start with an open list, representing nodes we have yet to visit and a closed list, representing visited nodes (nodes for which f(x) have already been calculated). The closed list is empty, and the open list contains only the starting node. The entire algorithm happens in a loop which continues so long as the open list has nodes in it. If the list is empty and we break out of the loop for this reason, it means we have failed to find a solution.

For each iteration of the loop, several things must be done:

1. The node with the lowest f(x) in the open list is our current node
2. If the current node is the end node we're looking for, we're done -- break out of the loop and repeatedly follow nodes' parents to see the path we found
3. Move the current node from the open list to the closed list
4. For each neighboring node to the current node, do the following:
    a. Calculate the neighbor's g(x) and h(x), and therefore f(x)
    b. If the neighbor is in the open list and the current g(x) cost is better than the previous g(x) cost, do the following:
        i. Save f(x), g(x), and h(x) to the neighbor
        ii. Set the neighbor's parent to the current node
    c. If the neighbor is not in the open list, do the following:
        i. Save f(x), g(x), and h(x) to the neighbor
        ii. Set the neighbor's parent to the current node
        iii. Add the neighbor to the open list

Because our goal was to use A* to compute paths between *multiple* objectives, and this approach assumes there is only one, we had to make some modifications to this template. To keep things simple, a list of goals is kept outside of the loop shown above, and those goals are precomputed using heuristics to estimate the optimal path between them. Now, this has some obvious problems – we can't *actually know* the optimal path between the goals without using A* algorithm between each goal and every other goal, but this is extremely slow for many objectives.

To resolve some of the inaccuracies of our faster approach, we integrated a check in the algorithm that looks for objectives that were not the originally intended objective. If such an objective was found (usually because there was an obstacle in the expected path), the algorithm resets and recalculates the optimal path using the same start node but *replacing the objective node with the one that was found by happenstance*. This affects performance minimally, unlike running A* between every goal, but results in paths that are much less self-defeating than if the algorithm is run without the check.

Finally, we had to account for objectives being clustered together, which we

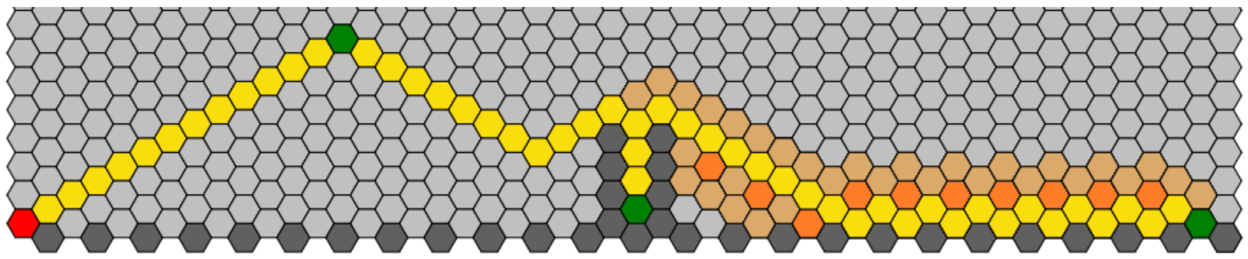
Using the Javascript App

Before we move on to the next section, where we test the A* implementation of the Javascript applet, we will make it clear how to use the app. Upon opening the webpage, you should see a grid of hexagons. Clicking or right clicking any hexagon when there is not a starting node will result in one being placed. If a starting node is already placed, left clicking will place obstacles and right clicking will place objectives. Clicking the "Run" button will cause the algorithm to dynamically find a path between the starting node and all objectives. The "Clear board" button can be used to start over with an empty board, and the slider below the buttons can be used to change the speed at which the demonstration executes (all the way to the left = instant, all the way to the right = one node evaluation per second).

Performance of Different Heuristics

To evaluate the performance of different heuristics in the A* algorithm, we used a fairly simple metric. For each heuristic, we simply checked how many nodes were in the closed list (how many nodes were evaluated/visited during the algorithm's execution) upon finding the optimal path.

We ran several heuristics on the test case shown below. Below the test case is a table with the results of the tests we ran, with some details on why the heuristic might be used.

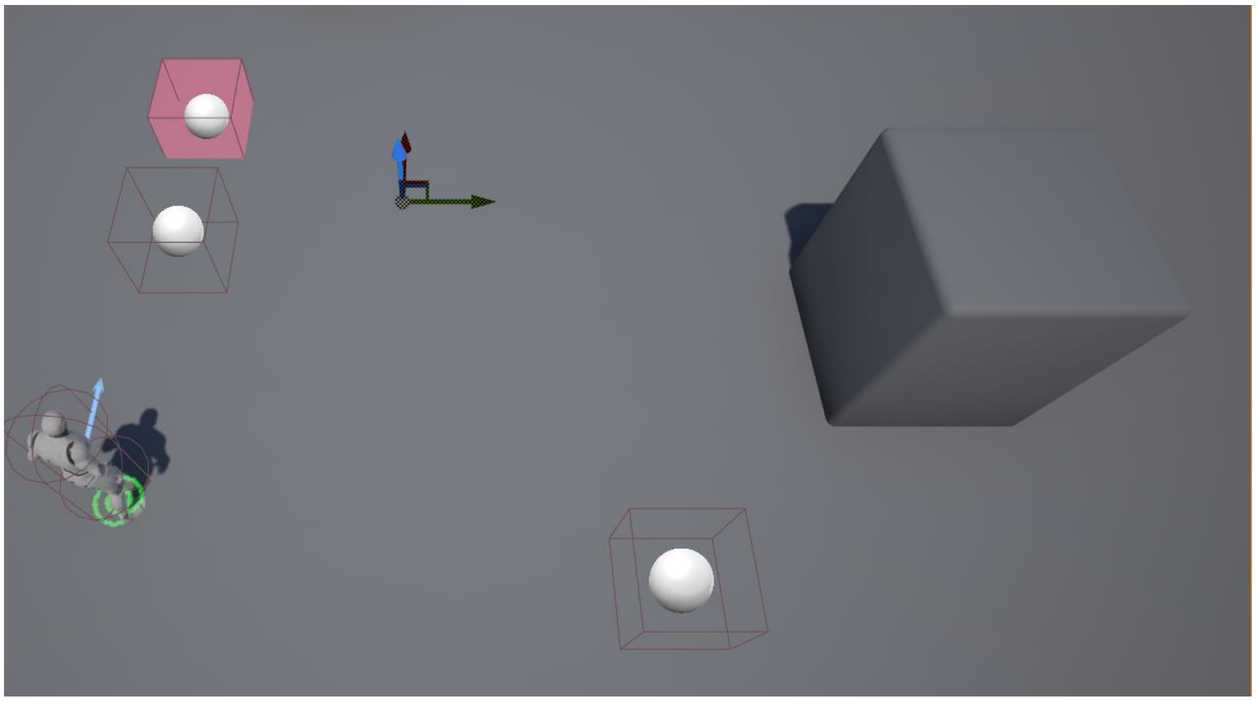


Heuristic	Nodes Visited	Advantages
Manhattan Distance	91	Simple, fast, usually finds best path
Diagonal Shortcut	110	Always finds best path
True Distance	120	?

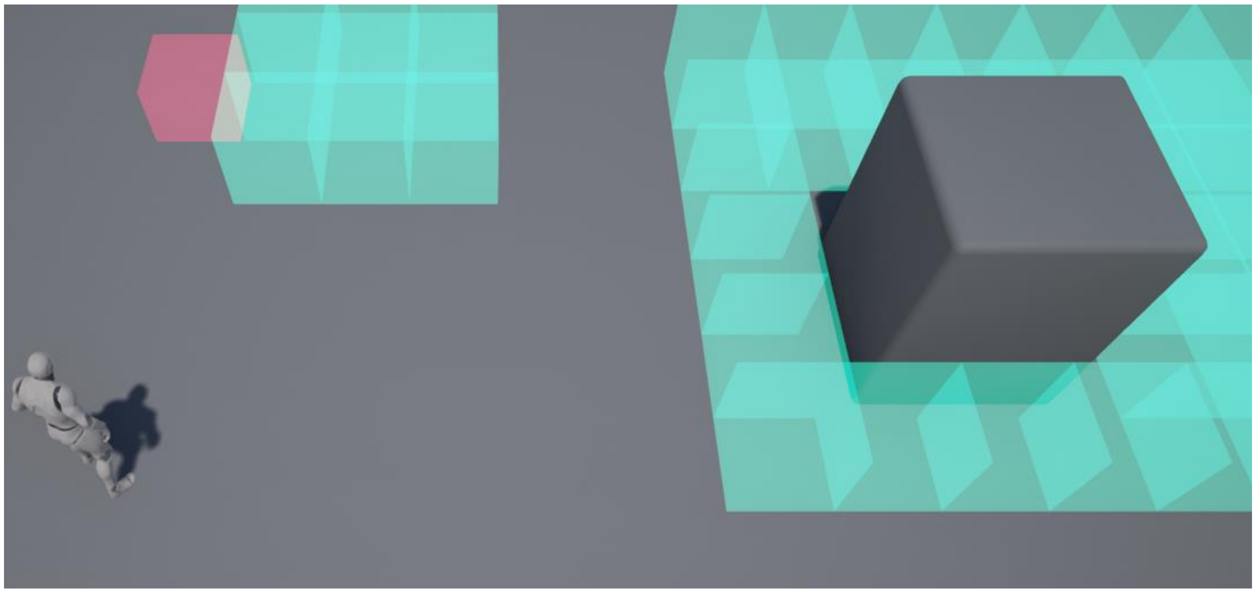
Clearly there is a decision to be made when choosing a heuristic between speed and pathfinding – what's interesting is that there would be no need for this tradeoff to be static. It would be entirely feasible to allocate more effort into pathfinding by using something like diagonal shortcut if the CPU is free, but then fall back to Manhattan distance when the CPU is doing a difficult task. It likely would be difficult or impossible to make this switch while a particular algorithm was running for a solution between two *nodes*, but trivial to accomplish between waypoints.

A* Unreal Modification:

Unreal Engine 4 uses a type of A* as their algorithm for movement calculations. However, unlike some A* implementations Unreal uses a navigation mesh, which is calculated when the game environment is build and provides precalculated fast paths between points of interest. The navigation mesh being built at build time of the environment does not make it very dynamic, and it has trouble reacting to a shifting world. My objective was to modify how Unreal functions to be able to modify movement calculations on the fly, a task which I was only partially successful in.



The above photo shows the game world as it is at build time. This is before the game is running, essentially a representation as to what the game looks like within the editor. You'll notice just a couple of boxes, a player character, and a more solid looking environment block. The following photo shows the game after it has begun running.



You'll notice that many blocks have materialized, but they do not intersect with the environment. These blocks represent modifications that have been made, at runtime, to Unreal's A* navigation map. The blue blocks represent a space that a character will try to avoid if they can, but if it is significantly more efficient to walk through the blocks, then they will. The red block (the color is scaled based on priority relative to other modifications that were made) represents an area that characters will more or less never walk through unless told to enter that space directly.

Using this method, the navigation mesh can be modified at runtime. If the player was given the power to spawn these blocks, then the navigation mesh would change as they were placed. This is *almost* what I had set out to do initially. Unfortunately, I cannot specify the intensity of the modification at runtime, I must make blocks with a specified modification for the navigation mesh beforehand, and the construct those at runtime. To be able to specify a dynamic value of navigation mesh modification I would have to have a more complete knowledge of Unreal engine, specifically the C++ side of it, which I was not able to grasp over the course of the project.

Future Goals:

- Adding "difficult terrain" to the Javascript app (this should not be very hard)
- Adding clustering to the Javascipt app – this was attempted, but never completed! The goal was to use hierarchical clustering to find clusters of objectives, and prioritize them over lone objectives