A* Pathfinding Algorithm Visualization

Final Report

AUCSC 310

Oscar Jaimes

Introduction:

The A* Search Algorithm is a variant of Dijkstra's algorithm which is a graph traversal and path search algorithm. A* is a best-first search algorithm which means the algorithm explores a weighted graph by expanding the best valued node chosen according to a pre-determined function that outputs a certain cost for the node in order to reach the goal node in the graph.

The main purpose of this project was to construct a visualization of A* such that a user would be able to choose between different heuristic functions and be able to easily place a start node, end node, and walls in the visual representation of a graph. In order to accomplish this goal, the visualization program was created in JavaScript as it can run on a web browser and provide ease of use as it serves as a solid medium for a visualization.

Before the implementation and visualization of A* took place, a better grasp of the concepts that accompany the algorithm were needed.

A Brief Explanation Of A*:

Since A* is a best-first search algorithm, it starts by looking at the start node, and then traverses to the next node with the lowest cost. During each iteration, the algorithm computes three functions for the adjacent nodes of the current node being expanded. These functions are *g(n)* - the cost from the start node to a given node, *h(n)* - the heuristic function which returns an informative guess of the distance from the current node to the end node, and *f(n)* - the sum of *g(n)* and *h(n)*. The algorithm stores all expanded nodes in two minimum priority queues also know as minimum heaps. The nodes that have already been expanded are enclosed in the closed list, and the nodes that have not been expanded are enclosed in the open list. Since both of these lists are minimum priority queues, at each iteration the algorithm extracts the node with the minimum *f(n)* value from the open list. The algorithm repeats this until the end node has been reached.

Algorithm Components:

For my implementation, I chose to take an object-oriented approach that was composed of five main classes. These classes were: Node, Map, PriorityQueue, Search, and Sketch.

Sketch Class (Sketch.js)

The Sketch class is a native P5.js class that usually implements two functions: setup() and draw(). It was used to place the graphical components on the screen and render nodes in case they were clicked, or their color changed.

Node Class (Node.js)

In regard to the Node class, I wrote a constructor that initialized ten instance variables: xPosition, yPosition, startNode, endNode, barrierNode, color, gCost, hCost, fCost, and parent. This class also contained seven static variables that included: startNodeX, startNodeY, endNodeX, endNodeY, nodeArray (a one-dimensional representation of the nodes in the graph), startNodeExists, and endNodeExists. This class also contains numerous functions such as getters and setters as well as a function that resets all the instance attributes of all nodes on the graph.

Map Class (Map.js)

For the map class, I chose to convert the previously made one-dimensional array of nodes (Node.nodeArr) into a two-dimensional array as this class was a representation of an adjacency matrix for the nodes in the grid. Although not a traditional adjacency matrix, the map was structured in a way where the adjacent nodes of a given node could be found seamlessly via a getChildren() function. The map class is the foundation for the search class, which will be covered in the next section.

Search Class (Search.js)

The search class can be said to be the most important part of this implementation as it is the class in which the A* algorithm is contained in (and it is the class which produced the most bugs). This class also computes the *g(n)*, *h(n)*, and *f(n)* values for the nodes that A* traverses. The constructor of this class takes in a map, a start node, and an end node as arguments.

Priority Queue Class (PriorityQueue.js)

This class was a simple representation of a minimum priority queue (a minimum binary heap). This class was solely made for the implementation of the closed and open lists that A* uses to keep track of the nodes it has traversed.

The Importance of Heuristics

While developing this visualization and researching the A* algorithm, it became very clear how important heuristic functions are for computing science and mathematics. Heuristics can be applied to many algorithms and are heavily used in artificial intelligence. There is a large array of computing science problems, such as NP hard problems, that are known to have a polynomial time complexity with the algorithms available today. Heuristics can be very helpful for these sets of problems as they can provide a good enough solution in a reasonable time frame for the problem. Although the solution may not be perfect, it is still valuable as it takes a reasonable amount of time to obtain a result.

For this implementation, I chose two common heuristics used in two-dimensional grids to be chosen by the user of the program. The first heuristic implemented was the Euclidean heuristic, which calculates the hypotenuse of the right triangle created by plotting the *x* and *y* coordinates of a given node and a start node on a two-dimensional plane. The second heuristic implemented was the Manhattan heuristic, which calculates the sum of the vertical and horizontal distance from a given node to the end node.

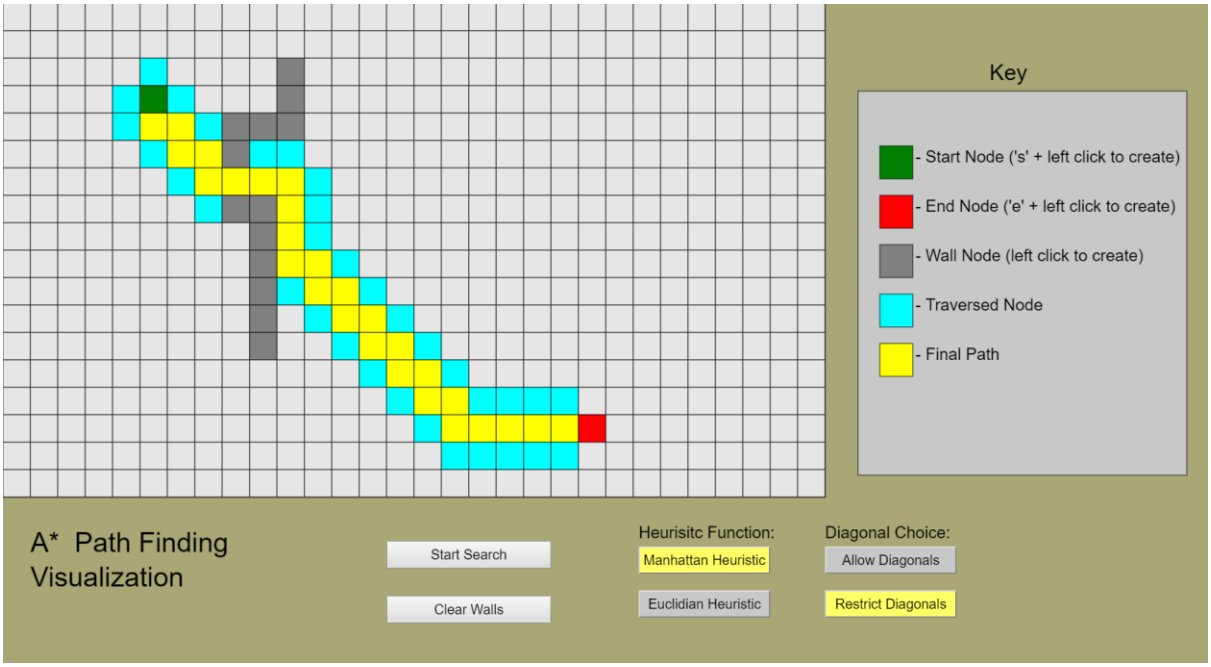
Time and Space Complexity:

Through analyzing and researching the A* pathfinding algorithm, many sources concluded that the running time of the algorithm is dependent on the heuristic function used. Although this is indeed the case, one can also analyze the time complexity as follows. First of all, as A* is searching for the goal node, it adds all the traversed nodes to the closed list; and when the goal node is found, the algorithm terminates. Therefore, the nodes stored in the closed list at the termination of the algorithm can be thought of as a subgraph of the original graph such that $|V|$(The cardinality or size of the set of nodes) is b^d where b is the branching factor of the graph created and d is the depth of the goal node. Let's assume that we preform a series of operations on the nodes, x. The running time of the algorithm would be $O(x(b^d))$.

Space complexity of the A* algorithm is simpler to comprehend. Since the algorithm stores all expanded nodes in priority queues, the space complexity of the algorithm would be $O(b^d)$ where b is the branching factor of the subgraph of nodes and d is the depth of the goal node (as explained above).

User Interface:

As the main purpose of this project was to construct a visualization of the A* algorithm, the looks and design of the UI were not held at the highest priority. The program consists of a set of buttons with their actions displayed on them as well as a key for the user to understand the functionality of the program (see below for screenshot). The user interface can seem outdated, but it portrays the functionality of the program in a concise but understandable manner.



Conclusion:

Overall, this project was challenging and helped me further my understanding on various topics in computing science such as algorithm analysis, heuristic functions, and real-life applications of graph traversal algorithms. Additionally, this project helped me combine my knowledge of software engineering and theoretical computing science and definitely strengthened my comprehension on research topics and content shown in class. The entirety of the source code for this project is hosted on GitHub(https://github.com/osc-jaimes/A-Visualization) and can be run by downloading the source code and dragging the index.html file into a browser, preferably Google Chrome.