Pathfinding Algorithm Research

Questions

1. What methods are there to find the path from one point to another?
2. Which method is more efficient than the rest?
3. Why did we decide to scrap the pathfinding function and choose the method of preset paths over it?

MAIN: Which pathfinding algorithm is best for our project?

Contents

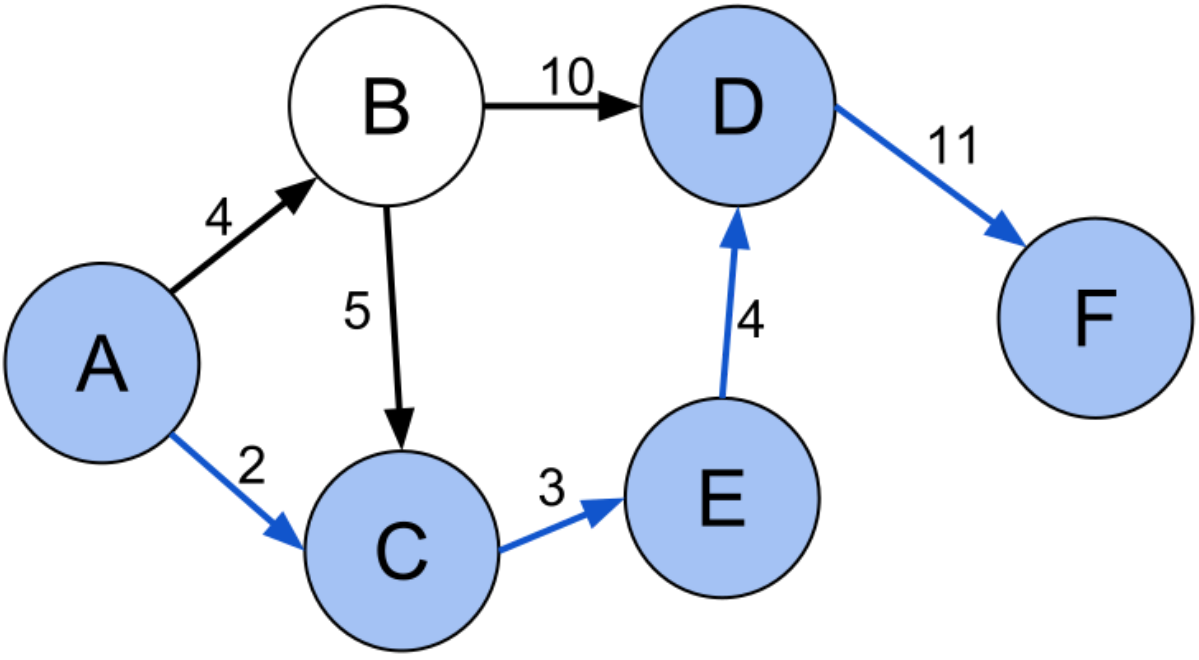
Introduction

The purpose of this project is to archive our findings in the topic of pathfinding and answer common questions asked by stakeholders and team members alike, which can be found above the contents of the document. This paper is not made to be a detailed explanation of pathfinding algorithms in programming and thus will not display full tutorials on how to create them in code form - only explain the general process of creating the algorithm, hence there will still be worded commentary with examples for stakeholders and PO's.

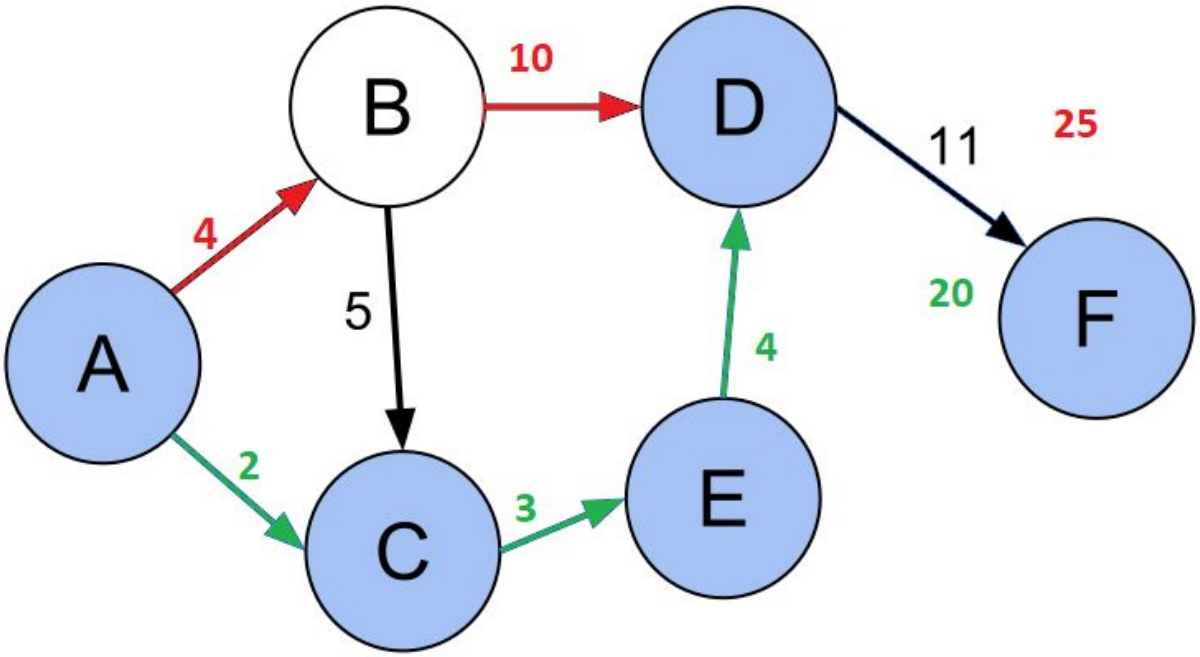
What is a Grid System?

Before we delve into the structure and idea behind pathfinding algorithms, we must first understand what environment they work in, so that we understand their functionality more in future paragraphs.

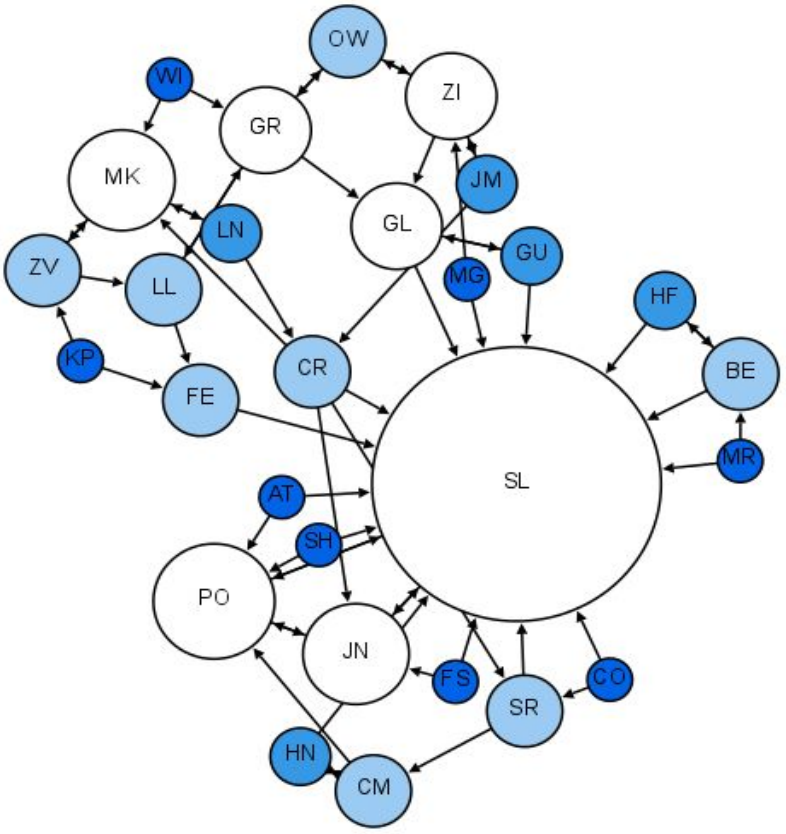
A grid system is a mathematical structure with multiple objects, connected by relationships in pairs. The general naming for the objects is a node, verticle or point and for the relationship connections is an edge, link or line. For simplicity, for the rest of the document we will use nodes for objects and links for connections.



This is a very simple example of a grid system with only 6 nodes and 7 connections. The numbers on the connections hold a value, making this a weighted grid - where traversing through connections has a different value depending on the path, for example, A would take to F. Let's imagine the grid system is actually a map, and the nodes are crossroads or key locations and the weight to the connections is the length between the connected locations. You could take the route from A to B to D to F, but that will hand you a sum value of 25. Meanwhile the route of A>C>E>D>F holds a value of only 20, making it shorter.



While this does seem easy to calculate just with the human mind, it becomes an entirely different story if the system looks something less than a simple math problem and starts looking a bit more like this:



Which is exactly why pathfinding is a necessary aspect when dealing with complex structures like these, as it is an algorithm that is easily applicable, but requires too many calculations for the human mind to take into account once the scale of the structure becomes larger.

Methods of Pathfinding

Now that we have arrived at the methodology of the pathfinding algorithms, it is important to understand how they work, before implementing them into your project.

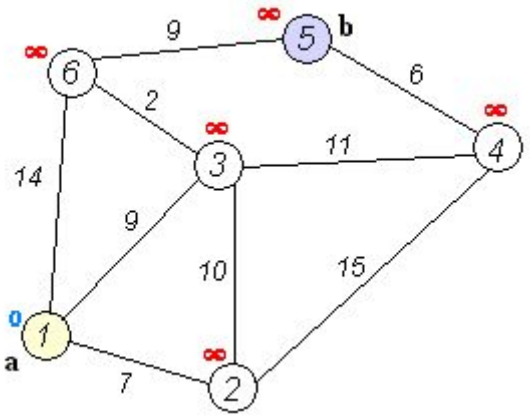
Pathfinding is the plotting of the shortest route between two points. Examples of such problems include transit planning, telephone traffic and in our case it is the beginning and endpoint of the route a user can take. The very basics of a pathfinding method is searching a graph, or a grid by starting at one node and exploring adjacent nodes until the destination node is reached, usually with the intention of finding the fastest route.

Within our research of pathfinding algorithms we encountered a few methods that could prove useful in the solution of our project.

Dijkstra's Algorithm

One of the more famous algorithms for pathfinding is one made by Dutch computer scientist Esdger Dijkstra. It follows the breadth-first principle of traversing data structures, where the routing algorithm search begins from the root or start node and traverses through its neighbours, and then follows those neighbours' connected nodes at a lower depth and recursively continues this process until it reaches the end node, then creates a path.

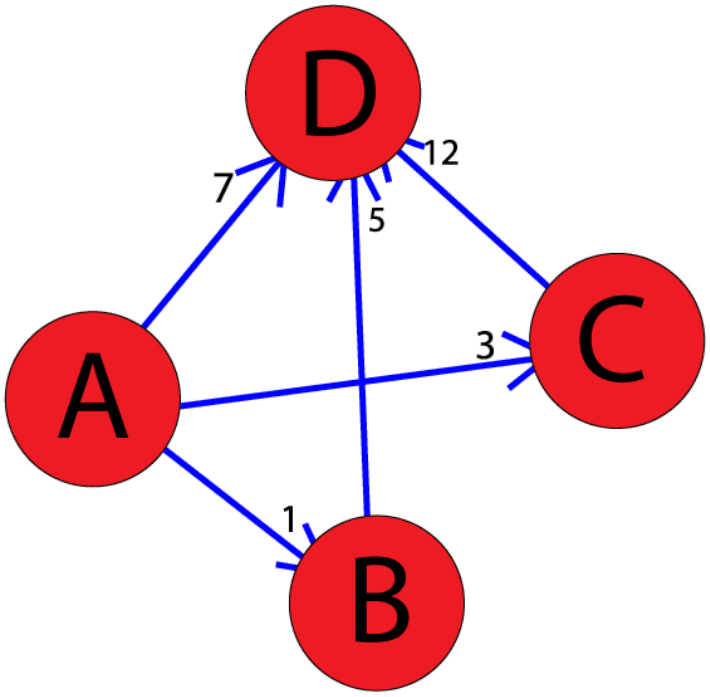
The main difference between breadth-first and Dijkstra is that when breadth-first is applied to an algorithm - it will not find the shortest path, which is what pathfinding in our product was intended for, instead it will display the first instance of a path found, which is not always the shortest. For instance, if we use the first grid in this document once again as an example, breadth first would show the A>B>D>F path as it is fewer objects than the optimal route. Dijkstra's algorithm takes this into account and with each check on a neighbouring node it calculates the weight of the connection it passes through. This way if a node is passed through twice it will be able to know which route is shorter, something that is not applicable in breadth-first.



A* (A Star) Algorithm

The A* is a variant of Dijkstra's algorithm and is a generic search algorithm that can be used to find solutions for many problems, including pathfinding. The A* algorithm is a breadth-first search algorithm that searches for shorter paths first instead of the longer paths. A* is optimal as well as a complete algorithm. Optimal meaning it is sure to find the least costing option in terms of time or space from the starting point to the destination. A complete algorithm means that it is going to find all paths that are available to us from the starting point to the destination.

The A* algorithm works with the following formula: f(n) = g(n) + h(n)
g(n) being the value of the shortest path from the starting node to node n, and h(n) being a heuristic approximation of the node's value. A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution. It trades accuracy for speed.



The idea here is that you start at node A and look at the connecting nodes, B, C and D. We want to calculate the lowest cost path from node A to node D. A > D is worth 7, A > C > D is worth 15 and A > B > D is worth 6. The optimal path from node A to node B, found using A* is A > B > D.

D* (D Star) Algorithm

D* is an algorithm that closely follows the principles of A* but with the ability to take dynamic maps and changing maps into account. Thus it is useful in the sphere of robotics, where one robot has limited data at first, which grows with time as its sensors gain more data.

Unlike Dijkstra where a node can either be marked or unmarked, nodes in D* have multiple possible values:

- NEW, meaning it has never been placed on the OPEN list
- OPEN, meaning it is currently on the OPEN list
- CLOSED, meaning it is no longer on the OPEN list
- RAISE, indicating its cost is higher than the last time it was on the OPEN list
- LOWER, indicating its cost is lower than the last time it was on the OPEN list

D* algorithm begins by searching backwards from the goal node, as opposed to Dijkstra and A* searching from the start. Neighbouring nodes are explored the same way as the other algorithms (but this time in reverse) and the calculations are finished once the start node is discovered.

The reasoning for this approach being effective in dynamic situations is the adapting it offers when met with obstacles in the predicted path to the start node.

“When an obstruction is detected along the intended path, all the points that are affected are again placed on the OPEN list, this time marked RAISE. Before a RAISED node increases in cost, however, the algorithm checks its neighbors and examines whether it can reduce the node's cost. If not, the RAISE state is propagated to all of the nodes' descendants, that is, nodes which have backpointers to it. These nodes are then evaluated, and the RAISE state is passed on, forming a wave. When a RAISED node can be reduced, its backpointer is updated, and passes the LOWER state to its neighbors. These waves of RAISE and LOWER states are the heart of D*.

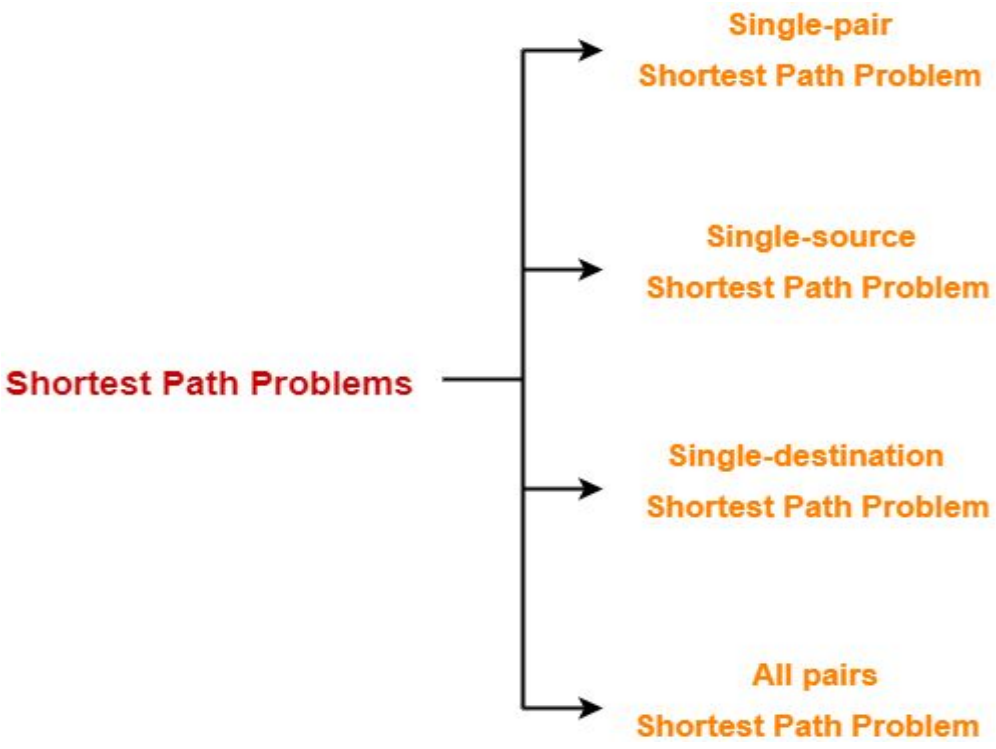
By this point, a whole series of other points are prevented from being "touched" by the waves. The algorithm has therefore only worked on the points which are affected by change of cost.” - D*, Wikipedia

Comparison of Main Methods

The following paragraph is made to answer question 2 of this research paper for the pathfinding algorithm that is best for our purposes. Each of them will be compared based on efficiency, as in the combination of the possible load the algorithm will have on the product and the success rate of given tasks, and complexity, how hard these algorithms will be to implement in a code environment.

Shortest Path Problem(s)

The Shortest Path Problem is what pathfinding algorithms are tasked to solve. It is a self explanatory problem, as it elaborates on the problem of finding the shortest or least-costly path between two objects in a data structure. But along with the SPP, there are multiple subtypes of it that specify the problem in more detail and context:



- Single-pair Shortest Path Problem
    - A problem where the pair of nodes is preemptively given before the start of the algorithm calculation process.
    - A* is the most commonly used method to deal with a Single-pair SPP, closely followed by Dijkstra.
- Single-source Shortest Path Problem
    - A problem which requires the shortest path from the start to all other nodes as a solution.
    - Dijkstra is a very commonly used method for this type of problem.

- Single-destination Shortest Path Problem
    - A problem which computes the shortest path from multiple nodes to the end node.
    - Dijkstra and especially D* are efficient solutions for Single-destination SPP.

- All Pairs Shortest Path Problem
    - A problem where the shortest path of each possible pair is computed.
    - Floyd-Warshall Algorithm and Johnson's Algorithm are the famous algorithms used for solving APSPP.

Let us look into the specific issue that our product holds. Guidance group is tasked to create a pathfinding app that shows the shortest path from a chosen start(root) node and an end node in a grid system. The design in mind is that the user may select one node to start the path and one node to end it, making our problem specifically a Single-pair SPP.

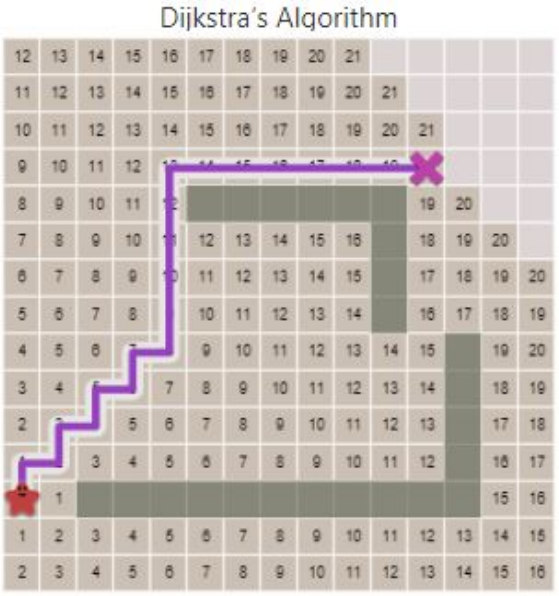
Hence, following the laws of the Shortest Path Problem, it is best to focus on either A* or Dijkstra.

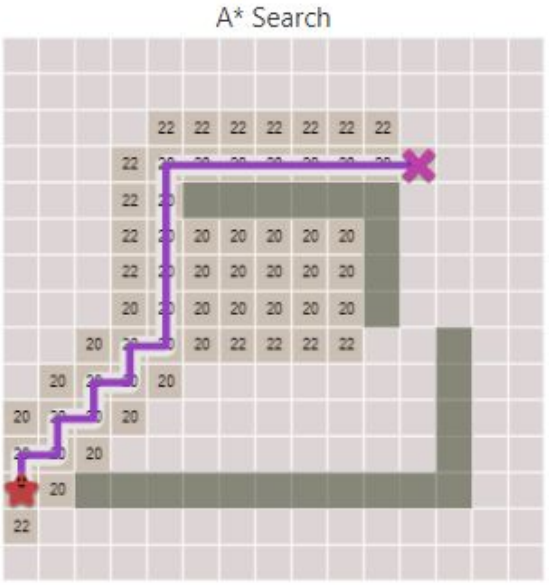
Efficiency

When using a pathfinding algorithm it is important to think about efficiency, if your algorithm isn't efficient, you'll either have a slow program or a program that does not work at all. The problem here is that with efficiency, comes complexity. In this case efficiency means the least amount of execution time and memory usage possible but still output the correct answer.

As the D* algorithm is mainly used for mobile robots and autonomous vehicle navigation; it is not the most applicable for our application so we will not be comparing it with the other algorithms.

The main difference between the A* and Dijkstra algorithm is that the A* algorithm is optimized for a single destination. Dijkstra's algorithm can find paths to multiple locations while A* finds paths to one location. Furthermore it prioritizes paths that seem to be leading closer to the goal. Although the Dijkstra algorithm lets us prioritize which paths to explore, so we can encourage it to use roads, instead of forests, comparing it to the A* algorithm which is optimized for a single destination it is clear that the A* is more efficient. Looking at the figure below you can see that both of the algorithms get the correct path, but the efficiency is different. Dijkstra explores more paths while A* only explored a couple.





Complexity

Besides efficiency, the complexity of an algorithm plays a big role when deciding which algorithm you want to implement. When we talk about complexity, the difficulty of implementing the algorithm is assumed, normally when you talk about the "complexity" of an algorithm people assume that it is the measure of the amount of time and/or space required by the algorithm for an input of a given size.

If you look at the formula of the current algorithms we can see that Dijkstra has f(n) = g(n), it means that it only looks for the path that minimizes the cost to reach an unexplored node directly connected to the nodes already visited, this making the algorithm fairly simple compared to the others.

When looking at the A* algorithm it uses a f(n) = g(n) + h(n) formula, all parts are the same as the Dijkstra algorithm except for the h(n) part of the formula. Here h(n) is an admissible heuristic function. An admissible heuristic is a heuristic ("a technique designed for solving a problem more quickly when classic methods are too slow"[1]) which never overestimates the cost of reaching the goal; the cost to reach a goal from a node is smaller than the optimal cost to reach a goal from a node.

After looking into the D* algorithm we found out that it is considered a top grade algorithm, which is hard to understand and to extend. Besides these findings there also is not a lot of documentation about it that is easy for us to figure out.

[1] https://en.wikipedia.org/wiki/Heuristic_(computer_science)

Conclusion to Comparison

In conclusion to this main paragraph, we believe that the best algorithm to fit our needs for our product is Dijkstra's algorithm. The reasons are several.

Firstly, it perfectly fits our needs discovered from the SPP research, as it and A* are the best for our type of task.

Secondly, you are most likely asking: "Well, why not A*? The research on efficiency shows that it is much better for single-pair pathfinding." and you would be correct. A* has proven to be much more efficient in this field and if a development team has the resources to work in a similar instance, they definitely should choose A*, but as mentioned earlier, increased efficiency means more complexity - and this is where our issue with A* stems from. Due to time constraints, we have to take the simplest, yet still efficient plan to realise our features, and to strike that balance we decided to go with Dijkstra's algorithm.

Reasons for Discontinuing in the Project

As it has been said in our recent sprints, the pathfinding algorithm has become a discontinued feature in our project and instead we have decided to allow for preset paths to be created from the administrators of their respective floor plans.

Why is that? The overall reason that has led to our decision was a mixture of complexity, lack of time and the risk of a faulty execution of the algorithm.

Complexity is easy to explain. Even when we considered it when researching these pathfinding algorithms, and even decided on the easiest one - Dijkstra - it was still too complex of a challenge for us to make an efficient implementation in the code base within the limited time that we had. Furthermore, the detailed explanation of our feature is “to find the shortest path between a start node and end node with the least possible population met”. So in reality our algorithm had to take two completely different values into account when executing the calculations, making it much more complex than a generic Single Path Problem.

Another problem was time. For six months we had to create a project more complex than anything any member of our group has done before along with multiple difficult features such as triangulation - which was nothing short of an impossible heap of work. After a lengthy team discussion about our progress in each feature of our product, it seemed best to look for alternatives in the module this paper is written for.

And lastly, risk and quality assurance. While the researched pathfinding algorithms are proven to be efficient and faultless, it is common practice for code to be fully tested to the best of a development team’s abilities before deployment. A big problem is met when we consider how to deal with pathfinding tests, as it is not only an extended mathematical problem, but also one with thousands, if not millions of calculations if a data structure is large enough. With this we risk delivering a shoddy algorithm to the product owner, when we can instead deliver a feature that can be securely tested, like the proposed preset paths feature.

Conclusion

Pathfinding is a very expansive topic with many use cases and solutions to these use cases. This document is created to compact knowledge gained throughout the development process of our project and our conclusions gained from comparing them and applying them to the plan for our product.

First we have documented what a grid system is so that we understand the environment these data structure methods reside in. We have also taken a look at three complex algorithms - Dijkstra, A* and D* and how each operates in a short explanation.

Afterwards we have documented our process of comparison between them. In this process we took a look at the Shortest Path Problem and used it as reference to see which algorithm is best for which specific problem. This was done not only to fit the general norm, but it would be easier for us to find resources to work with for our project if we used the popular algorithm for our specific problem.

Concluding the document we have set our choice on Dijkstra due to time constraints making it difficult for us to use the overall more efficient alternative - A*.

Citations

Chopra, C. (2019, October 21). Pathfinding Algorithms. Retrieved December 21, 2020, from https://medium.com/swlh/pathfinding-algorithms-6c0d4febe8fd

Koenig, S., Likhachev, M., & Furcy, D. (2004, February 12). Lifelong Planning A*. Retrieved December 21, 2020, from https://www.sciencedirect.com/science/article/pii/S000437020300225X?via=ihub

The Focussed D* Algorithm for Real-Time Replanning. (1995). Retrieved December 21, 2020, from https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.8257

Gwyn, S. (2013, November 24). D-Star Pathfinding Algorithm. Retrieved December 21, 2020, from https://seangwyn.wordpress.com/pathfinding-research/d-star-pathfinding-algorithm/

Singhal, A. (2019, October 30). Akshay Singhal. Retrieved December 22, 2020, from https://www.gatevidyalay.com/shortest-path-algorithms-shortest-path-problems/

https://en.wikipedia.org/wiki/Heuristic_(computer_science)

Example graphs:

Dijkstra's algorithm. (2020, December 19). Retrieved December 21, 2020, from https://en.wikipedia.org/wiki/Dijkstra's_algorithm

Shortest path problem. (2020, December 01). Retrieved December 21, 2020, from https://en.wikipedia.org/wiki/Shortest_path_problem