Introduction

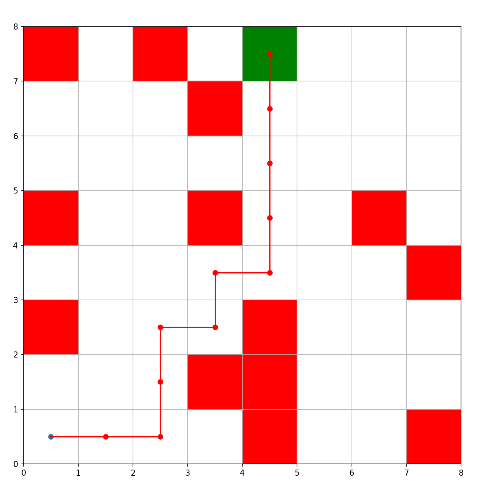
Pathfinding is crucial to the future of robotics. In order for a robot to complete any task, they need to know how to get from one location to another. Therefore, being able to calculate the shortest path between two points with obstacles in between is extremely important. There are many ways of finding the shortest path but each different method has their advantages and disadvantages depending on the scenario. For this research experiment, a grid was used simulate the field that a robot would move on. Certain squares were marked in red as obstacles where the robot would have to maneuver around. The different pathfinding methods tried, were Depth First Search, Breadth First Search, and Dijkstra’s Algorithm.

Abstract

Pathfinding is crucial to the future of robotics. In order for a robot to complete any task, they need to know how to get from one location to another. Therefore, being able to calculate the shortest path between two points with obstacles in between is extremely important. This paper examines the different techniques used to find the shortest path and then alters them to make them more effective through a machine learning model. After a set of experiments, it was found that to find the shortest path in a 2D grid, using Dijkstra’s algorithm was the most efficient. In a 64x64 grid with 400 obstacles, Dijkstra’s algorithm was on average 7.25 seconds faster than Depth First and Breadth First search. After adding heuristics to the Dijkstra’s model, it became faster but also became susceptible to traps and could often get stuck. In order to resolve that problem, a machine learning model was created. After experimenting, it was found that models with machine learning was on average 3.94 seconds faster than without machine learning. However, due to the simplicity of the model and the lack of training grids, accuracy decreased by an average of 18%. With greater resources and a stronger, more complex model it is possible to eliminate the loss of accuracy and retain the speed at which the model can find the shortest path.

Depth First Search

The first method used to find the shortest path was depth first search. In this method we would first find all the possible neighboring cells that we could travel to and store them in a stack. After this, we would travel to the most recently added cell in the stack, add the cell we visit into a visited array, remove it from the stack, and then repeat the process. What this does is that it travels down one path until it reaches a dead end. If the path does reach a dead end, we traverse back along the same path until we reach a cell where we can diverge from the path we just traveled. After we reach the end goal though, the algorithm doesn’t end. We first check if this pathway we followed was the shortest pathway we have found so far. It if is the shortest, we store the pathways in a list. If not, we disregard it and continue. We want to continue this process by traveling to the most recently added cell to the stack and repeat everything we just did. Once the stack is completely empty, it means that our algorithm has explored all of the possible options and the list we stored of the current shortest path, contains the actual shortest path. This solution does work, but it has some flaws. One of it’s biggest flaws is the time consumption that occurs. Since the graphs we work with are all cyclic, the number of possible paths become exponentially large, causing the program to run for long periods of time when trying to find the shortest path of larger grids. Cyclic graphs are graphs where you can travel back and forth from two cells; for example, you can go from Cell A to Cell B and then back to Cell A. However, with acyclic graphs, you can only travel in one direction, so we can go from Cell A to Cell B, but we can’t travel back to Cell A from Cell B. Overall, depth first search does give us the shortest pathway but is not feasible for any real world examples as the time it takes is too long and there are many better solutions.



Breadth First Search

The second method was Breadth First Search. In this method we would first start out by getting the neighbors of our current cell and then adding them to a queue. Instead of taking the most recently added cell, like in DFS, we take the first cell we added and travel to it. After we travel to that cell, we mark it in a visited array, remove it from the queue, and repeat the process. In Depth First Search, we travel on one pathway until we can’t go any further and then go to another pathway. But in Breadth First Search, we only move one spot in a pathway, and then go to another pathway. Once we hit the target, it actually means that we have found the shortest path. Using BFS, whatever path found the target is the shortest path you can take to reach it. This solution was a much more feasible option for finding a shortest path. The algorithm itself is less complicated and the runtime is very short as we don’t have to find all possible pathways.

A graph with red squares and green dots

Description automatically generated

Dijkstra’s Algorithm

Dijkstra’s Algorithm uses the same concepts as Breadth First Search but is used when the graph is weighted. A weighted graph is when traveling from one cell to another takes more than 1 unit. In a regular grid, traveling from on cell to another is just one space, meaning that grids are unweighted. In Dijkstra’s Algorithm, first add all the neighbors of our current cell to a queue, just like BFS. But instead of taking the first cell, we take the cell in the queue that has the shortest distance from the starting cell. Whenever we reach the target cell, the pathway we came through to reach it is the shortest pathway. Dijkstra’s and BFS are very similar algorithms, where BFS is used for unweighted graphs while Dijkstra’s is used for weighted graphs. Dijkstra’s is a much better algorithm in the real world as traveling to a certain place does not always take the same A graph with red and green squares

Description automatically generatedamount of time or energy.