A* Pathfinding in Dynamic Environments

Connor Blaha, Albaraa Alluahibi
dept. of comp sci

CS-44201
Kent State university
Kent, OH, USA

Cblaha1@kent.edu, aalluhai@kent.edu

I. Abstract

The project is created to implement an agent with path finding capabilities in a dynamic environment A* search. The agent moves in a grid that is created with user dynamically adding the start and goal, which resembles a real-world area like a warehouse or urban traffic system. We used A* search with Manhattan distance heuristic to inform the agent direction. For visualization, we use pygame to show the grid, the agent, itsdestination, and the dynamic obstacles it may encounter.

II. Introduction

One of the major problems with agents is pathfinding, because as we create the agent, we will need to make the agent determine which is the shortest and most efficient path to the goal while avoiding objects in real life or obstacles. These obstacles or objects may change place, or there will be differences in how the environment or map is structured from place to place. The goal of the agent is to navigate this environment of dynamic obstacles if given a destination in this environment. To address this problem, it is pertinent that the agent dynamically adapts to new environments or added obstacles whilst also choosing a path that isn't suboptimal. There are many algorithms chosen for pathfinding. Breath first search starts from a source node and bubbles out in an unweighted graph until it explores all nodes or reaches the destination. Dijkstra's algorithm does something similar, but it finds all optimal paths from the source node to all other nodes whilst considering weights. This is also computationally expensive. And the last of the famous pathfinding algori

This project focused on creating an agent that uses the A* algorithm for pathfinding in a simulated dynamic environment represented as a 2D grid with random or user-placed obstacles.

III. Related Work

A* has had a variety of applications. Its practicality has many applications from videogames, artificial intelligence, to satellite gps. There have been many variants of the algorithm that have various pros including better memory usage, better runtime efficiency, however sometimes these pros come with cons such as with better memory usage often meaning that the best path isn't guaranteed to be found.

One variant called Anytime Dynamic A*[1]. Anytime Dynamic A* addresses the problem of models in dynamic environments becoming quickly out of date. The idea is that when you have imperfect information and a dynamic environment, you need an efficient replanning algorithm to amend the previous solution developed from the imperfect information given. The issue with this is that sometimes developing this new solution can take longer than the agent must act in this new environment. The idea behind is to incrementally modify the solution as opposed to generating a new one given the current information. However sometimes it can be infeasible to generate an optimal solution within a time frame, thus the anytime part of the algorithm generates a solution that may not be the optimal solution but is best within the available computation time given. While our solution doesn't incrementally improve the solution like Anytime A*, it takes inspiration in the

sense that it updates the path as the environment changes so that the by feeding the new environment into a classic A* algorithm and produces new output. While this is admittedly slow, within the confines of our project this is deemed acceptable. If desired, the algorithm could be exp

IV. Methodology

This project implements A* to pathfinding within a dynamic grid environment that simulates the environment of a warehouse. It is developed using python and visualized using pygame. Our program consists of multiple parts.

1. The Grid
   The grid is the environment which the agent interacts with. The goal is for the agent to navigate the grid until it reaches the destination. It is customizable from within the python script with its` default size being 50 by 50 cells. Each cell can have a default value of either 0 or 1. A value of 0 means that the cell is an obstacle while the value of 1 means the cell is navigable. Our Algorithm takes a source point, and a destination point and this grid as input to output a path which will then be used by the visualization part of the program which occurs in the game loop.
2. Obstacle Generation
   There are two ways within our program that our obstacles can be generated. Randomly which generates random obstacles in approximately r percentage of the cells within the grid. Obstacles always generate such that a guaranteed path is guaranteed. The other way obstacles can be generated is by users painting the obstacles onto the grid like a canvas. When they do this, they can see the path update in real time and the agent adjust.
3. A* algorithm
   A* takes when used with grid-based pathfinding considers a 2-D matrix where each value represents a cell on the grid, 0(for an obstacle), and 1(for a navigable cell). It also accounts for information about each cell such as g(cost from the source node to current node), h(estimated cost to the destination node using a heuristic), f(cost which is the sum of g + h), parent_i and parent_j which corresponds to the coordinates of a parent cell which is where it is traversed from, this is used for path reconstruction later on.

   To check cells for validity, we have a method called is_valid() which checks if the cell lies within the grid boundaries, we also have a function called is_unblocked() which checks if a cell is not an obstacle. We have a method to determine if a cell is the destination cell called is_destination(), and a method to calculate heuristic code from the current cell called calculate_h_value().

   The algorithm first checks if source and destination cells are valid and unblocked. It initializes the 2-D list called Cell Details with stores the cost and parent information for each cell. A frontier min-heap is created so that cells are stored based on min_cost of f so that they are easy to retrieve for later. A list called reached is also initialized to track all cells that have already been visited.

   Lastly the source cell is initialized and stored in the frontier. The algorithm then begins the main loop of the algorithm. While the frontier is empty, the following steps are performed.
   a. Extract cell with lowest f value
   b. Market it as visited by putting it in reached list
   c. Check its neighbors in the directions up down left and right
   d. For each neighbor that is valid and not an obstacle, and not visited
      i. If it's the destination, reconstruct the path using backtracking and the cell_details 2-D list.
      ii. Otherwise update its g, h, f and parent coordinate values if it is not on the frontier, or the value the new f value is smaller than what exists already.

e. If a path isn't found, return an empty path which means failure to find a path

4. Game loop

A game loop generally works off taking the following events. First, process input.
Then use those inputs to then process the events. Once events are processed, display the output.

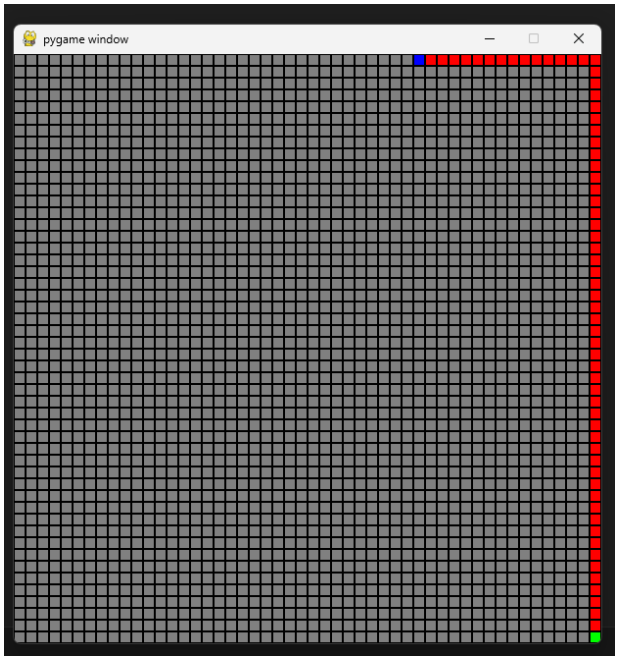
For the get input portion of the game loop, we check if the user wants to quit, if so quit the program, otherwise check if the user pressed o, r, b, or the mouse. If the user pressed o, generate random obstacles on the grid for our agent to navigate. If the user pressed r, remove all obstacles from the grid. If the user pressed b, enable drawing of obstacles or removal or obstacles. It works based off a boolean value that is simply inverted. If the user is pressing the mouse down, a boolean value is set to true and obstacles are either drawn or removed until the user releases the mouse.

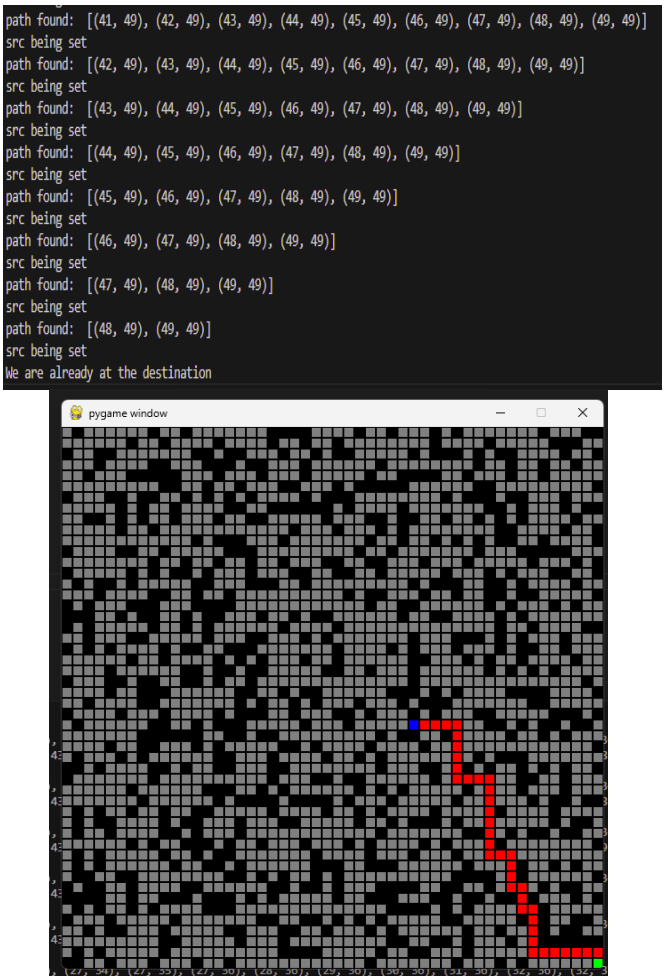
For the process events portion, we check if the mouse is pressed, if it is we get the position of the mouse and convert it to coordinates corresponding to places in the grid. If src is not set, we set the source, if src is set but we click it again, we are resetting and wanting to place the source somewhere else. If src exists but dest is not set, we set the dest. If both are placed and mouse is down, we draw a border so long as it is not the src or dest.

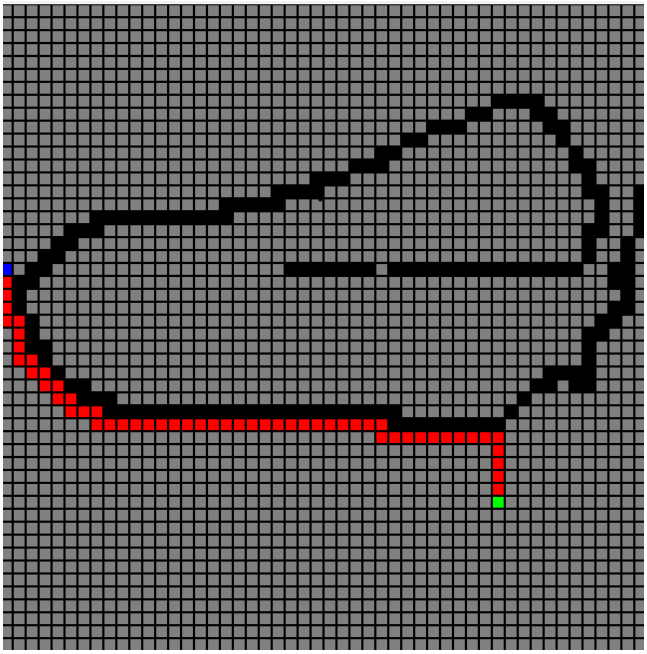
Lastly, outside of checking if the mouse is pressed, we check if the it is time to move the agent yet because we don't want the agent to zip across the grid so that we can't see the visualization.

V. Results

In the results of our implementation, we were able to run multiple pathfinding simulations using the grid. In each of them, we could add and remove obstacles. We also randomly generated an environment with obstacles using the function ranObstucal. Changing the gen value changes the number of obstacles generated in the grid, and anything equal to or above 0.5 will take some time until the agent finds a path to the goal. Also, using Pygame allowed us to have visualization for the user to see the agent moving toward the goal with the shortest path printed in the terminal. It also allowed the user to add obstacles, making the agent change to a new optimal path. Lastly, we used Manhattan distance heuristic to match the limitations of being able to move left, right, up, and down.







VI. Conclusion

In conclusion, the project shows how A* algorithm can be used for pathfinding in a dynamic environment using a 2D grid and real-time placement of obstacles, which creates a real-world environment to challenge our agent. In the future, we do not always recalculate the path when it is not needed would be beneficial. Another improvement would be optimizing our A* algorithm to make it respond faster or perform better at a larger scale.

References

[1] M Likhachev, D Ferguson, Geoff Gordon, Anythony Stentz, Sebastion Thrun “Anytime DYnamic A*: An Anytime, Replanning Algorithm,” in Proc. Int. Conf. Automated Planning and Scheduling (ICAPS), Monterey, CA, USA, JUN. 2005, pp. 262-271