CMSC421
Final Project
5-9-2024
Derek Miller, Tommy Chan, Akshay Subramaniam, Austin Shaw, Eric Detjen

AI Maze Solvers + Map-Terrain-Vehicle Solver

Literature Review:

Brkwok created a maze solver using JS, HTML, and CSS showcasing the DFS and BFS algorithms. We used this project as a starting point for the implementation of our BFS and DFS algorithms. Professor Justin Wyss-Gallifent gives an overview of Prim's algorithm as well as other search algorithms. We used these sources as guides for our maze generation algorithm and search algorithm implementations. Audi Victor Valenzuela from Medium gives an in depth explanation of how DFS can be applied to solving mazes. We used this source as a guide for our DFS implementation.

Introduction:

Maze solving is a very common problem in computer science, requiring algorithms to navigate through varying and random paths to find the optimal route from a start to a goal state. The key idea behind our project is to implement various algorithms which provide heuristics that will cause the AI to move in a certain direction. We offer the user the opportunity to simulate Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), Greedy Search, and Q-Learning to tackle maze navigation challenges.

Maze solving presents several challenges due to the maze's complexity and the sheer number of potential paths. The difficulty lies in finding an optimal solution given a heuristic, especially as maze size and complexity increase. Additionally, different strategies are required to address various maze characteristics such as dead ends, loops, and multiple goals. Each algorithm must be implemented correctly or we could run into problems such as the player never reaching the goal state, getting caught in an infinite loop, and more.

Efficient maze solving algorithms have real-world applications in various fields, including finding optimal paths in robotics, NPC navigation in gaming, routing in transportation, and network routing. By developing and comparing different strategies, we gain insights into their strengths and weaknesses, enabling us to choose the best algorithm for specific maze-solving scenarios. Ultimately, our project aims to provide a platform for exploring and understanding diverse AI techniques while addressing real-world challenges in maze navigation.

Main Concept:

The main idea of our project was to implement an automatic maze solver for randomly generated terrains using a variety of different search and training algorithms. We wanted to compare the performance of different algorithms in finding the most optimal path through the maze environment. Additionally, we wanted to see how the performance of these algorithms would be affected by the introduction of various different terrain types. The process of maze generation was difficult due to the variety of different cell types we incorporated into our project as well as the need to make the maze solvable at every generation. Additionally, the implementation of a q-learning algorithm in a randomly generated maze environment presented several challenges including: the complexity of environment dynamics, reward design, and algorithm tuning. It was also challenging to tune the algorithm for each vehicle such that they exhibited their expected behaviors.

Our application has the potential to be highly applicable in several real world scenarios. In robotics, robots often need to traverse through complex environments with diverse characteristics. By adapting our solution to this scenario, robots could potentially learn to better navigate through real-world environments. In the world of game design, maze-like environments are commonplace. Thus, user experience can be greatly enhanced by creating agents that can navigate these environments intelligently.

Initial Goals vs Results:

While we were able to meet most of our initial goals, there are a few that we did not accomplish. We succeeded in implementing maze generation with several different terrain types. We were also successful in implementing most of the algorithms we initially planned to implement. Additionally, we were able to incorporate different types of vehicles in our q-learning implementation. However, we were not able to have two algorithms running on the same maze simultaneously. Due to the difficulty of displaying two algorithms running simultaneously as well as time constraints, we decided that this feature was ultimately not worth pursuing. We were also unable to incorporate a fuel counter for vehicles and “fuel” tiles where vehicles could replenish their fuel supply. This was again due to time constraints and the difficulty of incorporating such a feature into our q-learning algorithm. Overall, however, we were able to accomplish most of our initial goals.

How Solution Works & How it was Developed:

For our project, we gave the user the option to either generate a simple maze that only included a start cell, goal cell, and wall cells, or a more complex environment that had all

different types of terrains: snow, water, grass, dessert, etc. We decided to implement 4 different search algorithms: BFS, DFS, UCS, and Greedy search (Manhattan and Euclidean), as well as q-learning. For the q-learning algorithm, we implemented different “vehicles” that would have different optimal paths depending on the terrain. For example, the boat would favor paths through water cells while the car would favor paths through dry land.

To start for BFS and DFS, we referenced Dr. Justin Wyss-Gallifent’s (of the University of Maryland) notes on BFS and DFS and implemented a solution to an algorithm based on his approach. Start by defining start and goal nodes, then create a visited array storing all the tiles that have been visited by the algorithm. Create a path queue (stack for DFS), which keeps track of paths of nodes to the goal. Use a recursive function to traverse through all the tiles, and for each function call pop the first tile off the queue (last tile off the stack for DFS) and add each of its adjacent tiles to the queue. We continuously check for empty queues/stacks, whether or not goal nodes have been reached, and whether or not a move is valid. When the first path finds the goal node, that path is then marked (path tiles are relabeled) in the maze 2D-Array as yellow and the display is altered to show this change, to show the quickest path to the goal node.

The greedy search algorithms we implemented along with UCS follow the same structure as BFS and DFS. They store potential nodes to visit in a priority queue, and the nodes are visited based on some heuristic (Manhattan distance from tile to goal node, euclidean distance from tile to goal node, previous cost plus one). For the Q-Learning algorithm, we trained on 15,000 epochs, at a learning rate of 0.1, discount factor of 0.9, and epsilon initially as 1.0 with a decay of *0.99. A Q-table is initialized with 0’s for each tile-action pair (can either move up, down, left or right). During training, each epoch starts at the initial state and while we haven’t reached the goal node, we either choose to take some random action or choose the best action to take from a certain tile, based on the current epsilon value, whether we want to explore or exploit. For each move taken, the reward is calculated and the q-value for a tile/action pair is updated. Once the model is trained, a priority queue is used to pick the nodes to visit next to find the fastest path. The heuristic value used is the action with the highest q-value for a tile. By following the max action values for a tile, the goal node will be found the quickest.

Q-Learning is also used to solve a map, based on the terrain of the landscape and the vehicle chosen. Based on the selection the user makes regarding the vehicle, the rewards for traversing through different tile types are altered. The Q-Learning algorithm remains essentially the same besides the rewards changing based on certain conditions. The map was generated as a 2D array, where tiles are both by random and by what preceding tiles are. There’s always a

40% chance a tile is the same type as the previous in the same row, 40% chance a tile is the same type as the previous in the same column, and a 20% chance the tile is a random type.

Strengths and Limitations:

Our solution is very user friendly and allows for a high degree of customization. Choosing between the different algorithms as well as changing the dimensions of the maze is easy and intuitive. Additionally, our implementation of the search algorithms shows the progress of the algorithm sequentially and highlights not only the final path, but also every cell that was visited by the algorithm. In contrast to simply displaying the end result without showing intermediate steps, this approach allows the user to easily see the algorithm's progress and explore how different maze configurations affect the performance of the algorithms. Our implementation of the q-learning algorithm offers the user a great deal of exploration potential by featuring a wide variety of terrains and granting the user the ability to choose between different vehicle types. This allows the user to more thoroughly explore how different environment dynamics can impact the performance of different modes of transportation.

Our solution has a few key limitations. Firstly, due to the randomly generated nature of the maze, the user is not able to fully customize the placement of walls and other cell types. Additionally, the start cell and goal cell are always in the same relative positions. This hinders the user's ability to experiment and explore how more interesting configurations could affect the performance of the search algorithms. If the user was granted the freedom to fully decide the environment of the maze, it would become much easier to test specific maze configurations that the user might find interesting. Another minor limitation is that there is no way for the user to run multiple algorithms at the same time. If such a feature were implemented, the user could more easily analyze and compare the performance of different algorithms.

Results/observations:

In the end we were able to create a fully functional maze solving application. The maze could generate mazes of different sizes (5x5, 9x9, 15x15, 25x25). We were able to implement fully functional algorithms such as breadth first search, depth first search, greedy search (euclidean and manhattan), uniform cost search, and q-learning. Each of these algorithms were able to fully traverse and find the unique path to reach the goal. We also had a biome/map generation that would generate a "real life" scenario where there are different terrains in the world. We had different modes of transportations such as boat, car, airplane, off-road truck,

snowmobile, and dune-buggy. There are different terrains such as sand, water, snow, grass, forest, and air. The q-learning algorithm suits this biome traversal best as it can avoid difficult terrains and choose beneficiary terrains when traversing.

Individual contributions:

Derek Miller: Worked with Akshay to help fully develop the Q Learning algorithm, made modifications to the Q Learning algorithm to account for different terrains and vehicles (fully implemented the map solver). Created formula to randomly generate maps to have tile types grouped into clusters. Worked on/ fine-tuned other algorithms (BFS, DFS, etc.) to make them work correctly. Fixed bugs in displaying paths as well.

Tommy Chan: Coded all of the buttons, event handlers, and everything in the GUI (HTML and CSS). Also used prim's algorithm to generate mazes of different sizes and displayed the mazes. Debugged various parts of the code and added reset functionality. Attempted to help with q-learning and double checked other traversal algorithms.

Austin Shaw: Coded the DFS algorithm and helped with debugging. Implemented helper functions to streamline the development of the search algorithms, most notably a function to check for valid moves. Test ran features of our solution including search algorithms and maze generation. Contributed to the final project report by writing multiple different sections.

Eric Detjen: Coded the BFS and worked on the other algorithms. Worked on the maze UI and the data structure storing all of the moves. In particular, I made it possible to see the entire path visited in addition to the final path.

Akshay Subramaniam: I worked on some of the Greedy Search - Manhattan Distance algorithm, some of the Greedy Search - Euclidean Distance algorithm, and the initial Q - Learning algorithm. I did the initial setup of these before running into a couple bugs that other teammates helped me fix.

Bibliography:

Brkwok, Brkwok. “Maze-Solver.” GitHub, GitHub Inc., github.com/brkwok/Maze-Solver. Accessed 9 May 2024.

Valenzuela, Audi Victor. “Solving Mazes with Depth-First Search.” Medium, The Startup, 9 June 2020, medium.com/swlh/solving-mazes-with-depth-first-search-e315771317ae.

Wyss-Gallifent, Justin (n.d.). https://www.math.umd.edu/~immortal/

Wyss-Gallifent, J. (2023, May). *CMSC 351: Spanning trees - UMD math*. CMSC351: \ Introduction.https://www.math.umd.edu/~immortal/CMSC351/notes/spanningtrees.pdf