Visualizing an optimized path finding algorithm

Alamgir Khan, Dhruvaa Saravanan, Aman Rana, Rahul Jaideep

April 2021

1 Introduction - A*maze-ing python pathfinding

- Problem Description:

  A maze is a network of paths and obstacles that culminate in a final destination. Traversing a maze is difficult because of dead-ends and obstacles. In our project we will be simplifying the layout of a maze into nodes and attempting to use a traversal algorithm to find the shortest route to complete the maze. By simplifying the junctions and paths of a maze into nodes, we will be able to use our knowledge of graphs to traverse the maze. Due to the scale of the maze, the shortest route will no longer be apparent to the human eye, which is where the algorithm will come in. We will try to visually show the maze and the proposed path that our algorithm will generate.

  After our research stage we felt as a group that the A* search algorithm would be the most efficient way to traverse our randomly generated mazes. The specifics of the algorithm we go into in the computational plan, but it works by minimizing a Cost function , f(x), so that the traversal is always in the direction of the end goal.

  Finding the shortest distance between two nodes has applications beyond maze traversal. The nodes could be used to represent junctions in a city, the algorithm would then show the route to a destination that passes through the fewest junctions. The nodes could be used to represent flight connections, the algorithm would then show the fewest number of connections between two destinations, potentially saving time and money for travelers. If expanded to social media it could show the degrees of separation between people on the platform and could be used to visualize the six degrees of separation theory (Wikipedia). The flexibility of the possible solution is the reason why we chose this goal. Although we are only creating the visualization for a maze, the actual traversal algorithm has many more uses and would only have to be slightly altered to represent different scenarios.

- Research Question:

  How can we determine and visualize the shortest route between two different points? For example, the shortest path out of a maze?

2 Datasets

Our project does not have any external datasets. However, our program generates its own random mazes. This is done in the create_random_grid function in visualization.py. This function returns a list of a list of Vertices. Each sublist contains 90 vertices and there are 90 sublists in the returned grid. Basically, each sublist represents a row in the maze and since the maze has a dimension of 90x90 nodes, there will be 90 such lists/rows.

3 Computational Overview

We created two files in our project that are called on by our main.py file; algorithm.py and visualisation.py. Below is an outline of the computation within these two files:

1. visualisation.py is our visualisation program that generates a maze and shows the traversal function to the user.

- Initially our main.py function starts by creating the pygame window on which our visualization will be based. We used the constant 900 to create a square window of height and width of 900 pixels.
- This window of hiehgt and width 900 is passed to visualisation.run_visualisation. That is, the window and its width is passed to the function run_visualisation which resides in the visualisation.py file.
- run_visualization is the main function of visualisation.py. The function starts by calling create_random_grid(), feeding it the WIDTH from before and the number of rows, which is a constant set to 90 at the start of the file. We explain the create_random_grid() function above, in the Datasets portion of this submission. The 'grid' created is really a list of lists of vertexes, that we eventually turn into a graph (as we explain later), which is then visualized as a grid on pygame.
- run_visualisation then sets a random node as the start position, and a random node as the end position, giving them the colours blue and purple respectively.
- The function then moves onto a for loop which runs so long as a running condition remains true. This condition is made false if the exit button is pressed on the pygame window.
- The function then calls the draw_grid function, a relatively straightforward function which draws the grid on pygame by calling another function called draw_grid_lines.
- Finally, we progress to the pygame event for loop, that waits for user inputs to start the actual pathfinding process. If 'm' is pressed, the pathfinding algorithm runs using the manhattan heuristic for the total cost function, if 'p' is pressed, the pythagorean heuristic is used instead, and the diagonal distance heuristic is used if 'd' is pressed. If c is pressed the grid is cleared of any progress the pathfinding algorithm made. Finally, if r is pressed, the grid is re-drawn with a new maze. I will explain how the pathfinding works and what the different heuristics are below:

2. algorithm.py is our program that contains the vertex class, which contains the update_neighbour initializer which helps us turn our list of Vertexes into a graph. algorithm.py also contains the pathfinding algorithm and the 3 different distance heuristics.

   - In our run_visualizer function as described before, once either 'm', 'p', or 'd' are pressed, a for loop is set in motion which calls to the Vertex.update_neighbour method, this is the part where the grid is transformed from just lists into a proper graph. In update_neighbour, each node checks around itself to see which nodes around itself are traversable, it then adds these vertices to its neighbours list, if there are any. This is done for every node in the grid list, making a graph which our pathfinding algorithm can use.
   - For our pathfinding algorithm, we came across multiple possible methods for traversing a graph, but settled on the A* search algorithm (Stanford). To choose the next node to move onto, our algorithm makes use of a total cost function. The total cost for each adjacent node is calculated, and the one with the lowest total cost is chosen as the next node to move to. I explain the total cost function below:
   - **The Total Cost Function** (which will now be known as f(x)) is the principle that will be governing our choices when traversing the grid. We will use the convention that f(x) = g(x) + h(x). g(x) is the number of nodes traversed so far, and h(x) is the heuristic that gives the shortest distance between the node being scrutinised and the end point. There are multiple ways to calculate the possible h(x), which is why we offer three different possibilities, the Manhattan heuristic, the Pythagorean heuristic and the diagonal distance heuristic. The closer the h(X) distance is to the real distance, the better our pathfinding algorithm will perform. By minimizing our f(x) at every traversal, we aim to find the adjacent node with the lowest h(x), this ensures that our traversal is always moving in the right direction, towards the endpoint. Since we need to know where the endpoint is to calculate h(x), the A* search algorithm is an 'informed' search algorithm.
   - The 3 heuristics we provide work as follows: (We used the Stanford theory website on the A* algorithm as a starting reference for these heuristics)
     - **The Manhattan heuristic** assumes only 4 directions of travel, up, down, left and right. It uses linear algebra to find the distance, projecting the line segments that make up the traversal onto the coordinate axes and summing the resulting lengths. (Wikipedia)
     - **The Diagonal Distance heuristic** is the most suited to our pathfinding algorithm, it assumes 8 directions of travel, and our program allows for diagonal travel. The heuristic is such that it assumes 8 directions of travel. It takes the number of steps from the Manhattan distance and subtracts the additional steps from allowing diagonal travel.

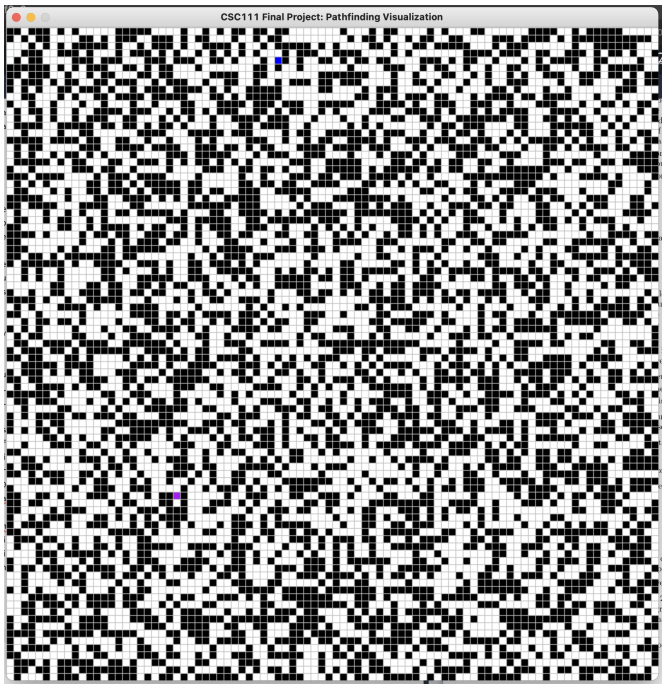
  - **The Pythagorean heuristic** finds what is known as the Euclidean distance, its the distance between two nodes, taken directly as the shortest distance between them, using Pythagoras. This is the least accurate of our methods, therefore ranks the lowest in terms of efficacy.

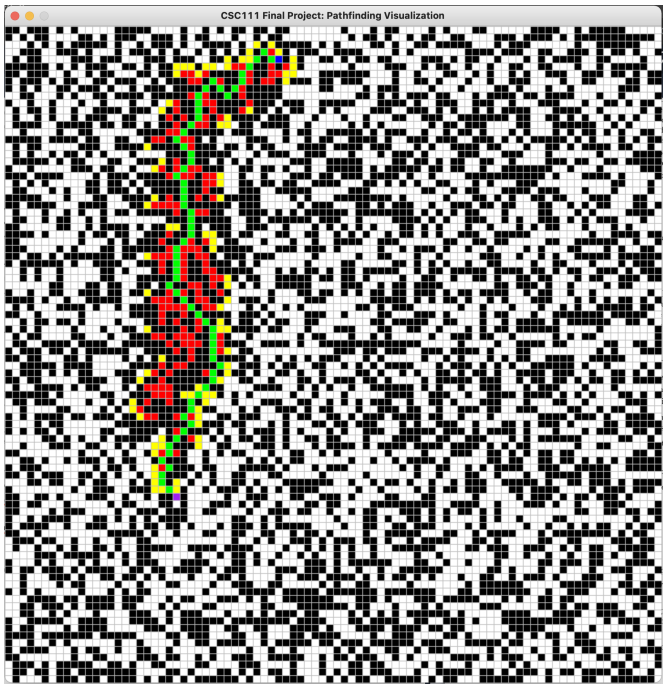
- The project requirements state that we must either use a new library or use Pygame in a new way. To that end, we implemented a function called make_path that works in tandem with our visualisation to show the pathfinding algorithm working in real-time, so the user can see how it considers nodes and always makes its way closer to the end goal. Additionally, we enabled user interaction, the user can generate new mazes at will, and try different heuristics similarly.

4 Instructions

- Only Python (version 3.9) is required.
- The libraries/modules we will be using are (all pure Python):
  - pytest
  - python-ta
  - pygame
- To run our project, only the main.py file needs to be run. When it is run, a pygame window should pop up with black and white squares. This is our maze map - the black squares represent walls and the white squares are traversable. You should also be able to see a single blue square and a single purple square. The goal of our program is to find the shortest possible route between the start point (blue square) to the end point (purple square).
- There are three heuristics that we have implemented in out program and either of these three can be used to find the shortest path. To recap, the three heuristics are the Manhattan, Pythagorean and Diagonal heuristics.
  - Press 'M' to use the Manhattan heuristic in the total cost function.
  - Press 'P' to use the Pythagorean heuristic in the total cost function.
  - Press 'D' to use the Diagonal heuristic in the total cost function.
- At the end, the shortest path found should be represented by green squares. In the case that no path is found, the program will end with every single searched square filled red.
- Once a path is found (or not), you can try a new heuristic or even clear the path and generate a new maze.
  - To clear all paths, press 'C'.
  - To generate a new random maze, press 'R'
- As long as all the modules in the requirements.txt file are installed, the file works perfectly in new projects on any new Windows or MacOS machines.

Below I have included the visualization of the initial state and the final state of the pygame window.





5 Evolution of Project

All in all, unlike our project for CSC110, we did not drastically change our project idea from our proposal. Based on the feedback from our TA on our project proposal, we decided to make a couple of changes.

- Firstly, we decided on a more descriptive title (it was initially Traversal/Pathfinding Algorithm)
- We were advised to use more than just 20 mazes and possibly generate them in our program. So, we decided to include a function (create_random_grid) that randomly generates mazes. This function can be found in the visualization.py file. Because our mazes are randomly generated and have a dimension of 90x90, we can generate 2^{8100} distinct mazes.
- We were also advised to look out for potential recursion errors. Our final implementation does not have any recursion and so we cannot get a recursion error. Our program instead uses a while loop to update the paths accordingly and if no path is found then the program stops.
- Finally, we specifically decided on implementing the A* search algorithm.

6 Discussion

- Our computational exploration definitely helped us achieve the research question we started with. Our goal was to see if we could create a process where the shortest distance between two points is found, rather than just any arbitrary traversal. Additionally, by creating a pygame visualisation where the user can watch the pathfinding algorithm snake its way to the goal, we are able to more intuitively understand how the algorithm reached its conclusion. If we simply displayed the final path, it would be much harder to understand the method behind the traversal.
- A limitation of our program is the way it handles the probability of an impossible traversal. Since the grid is randomly generated and the start/end nodes are randomly chosen, there is a possibility that the end node will be surrounded by obstacles in a way that makes it inaccessible. This is extremely unlikely since the grid is so large and we allow for diagonal traversal. However, in this possibility, our program keeps running until it explores ever accessible node. This means that our worst case running time is calculated as the running time to search the entire grid, which is quite long. If we had some way of identifying when a node is inaccessible, it would help lower our worst case running time and make it easier for an end user to use.

- Another limitation of our method is that it isn't very scalable. Increasing the size of the rows from 90 to 180 makes our method incredibly slow and it can be frustrating. Perhaps using recursion is some sequences rather than loops could be one way of improving our code as our implementation would simply be infeasible for real world GPS use.
- We are also limited by the nature of the A* search algorithm, it is an informed search, therefore requires the position of the end node and the position of every node in the graph relative to each other in order to work. In scenarios where we do not know where the final destination is, our algorithm is worthless, we would have to look into different search algorithms.
- To improve the implementation, we could adapt the program to traverse any graph, rather than just the hard-coded pygame maze. This would be truer to our project proposal, where we hypothesised that such algorithms could be used for GPS systems or other travel optimizations.
- Furthermore, to compare the strengths of different algorithms, we could try implementing a search algorithm other than the A* search algorithm. We know that there are other possibilities such as Dijkstra's algorithm, since A* is meant to be a refined version of Dijkstra's algorithm, we could compare to see just how much better it really is. And if different algorithms work better in different situations.
- The basis of our research was to understand path-finding and how algorithms determine the shortest route between 2 places which naturally leads to how GPS systems work in our daily lives. Whilst our work definitely took a step in the right direction, we would require far more advanced technological understandings of artificial intelligence, algorithms, decision making, recursion, traversal and so on, in order to fully scale, reflect and compare how pathfinding works in the real world.

7 References

- Wikipedia Contributors. "Six Degrees of Separation." Wikipedia, Wikimedia Foundation, 4 Mar. 2021, en.wikipedia.org/wiki/Six_degrees_of_separation. Accessed 16 Mar. 2021.
- Wikipedia Contributors. "Taxicab Geometry." Wikipedia, Wikimedia Foundation, 17 Feb. 2021, en.wikipedia.org/wiki/Taxicab_geometry. Accessed 15 Apr. 2021.
- "Heuristics." Stanford.edu, 2021, theory.stanford.edu/ amitp/GameProgramming/Heuristics.html. Accessed 15 Apr. 2021.