Artificial Intelligent Technique I: Software-Based Approaches

Maze Project Using Multiple Search Approaches

Prepared for

Aj. Patinya Ketthong

Prepared by

Shotika Okabe (Rei) 2021610130

Nakamon Katekaew (Opal) 2021610148

Manocha Tanrujirote (Jaonai) 2021610197

Juntorn Thiantanukij (Jamie) 2021610213

October 22, 2021

Content

Executive Summary

Purpose of this report

The Breadth-First Search, Depth-First Search, A* Search, Dijkstra algorithms were implemented into the pathfinding problem in a maze.

The purposes of this report are as follow:

1. Define how to implement search algorithm into pathfinding problem
2. Analyze each method efficiency and determine which one is the most optimal algorithm

Methods Used

All the search algorithms mentioned above will be done in python to visualize how programs search through each node and to display the route of each search.

Findings and Conclusions

A good routing algorithm should be able to find an optimal path and it must be simple. The shortest path problem is defined as that of finding a minimum-length (or cost) path between a given pair of nodes. The A* algorithm is considered the most efficient method for shortest path computation.

On the other hand, Depth-First-Search is not optimal for pathfinding. DFS uses Stack to find the shortest path. The number of steps (nodes) to reach the destination (solution) could be very high. It produces a much longer path. Whereas in Breadth-First-Search is more optimal with its tier search algorithm. BFS uses Queue to find the shortest path.

However, BFS is only optimal when nodes are unweighted. When there's a different weight assigned to each node, A* will find the shortest path between start and goal node whereas, Dijkstra can also find the shortest path but it wastes time exploring in directions that aren't promising.

Introduction

Pathfinding, or finding a route to a destination that avoids obstacles, is a classic problem in AI. This is the most common usage, and web mapping tools such as Google Maps, satellite navigation systems, routing packets over the internet. There's a lot of search algorithms but only a few can give the most optimal routes.

This project aims to determine the best search algorithm for pathfinding by using a maze to help visualize how the program finds its best route. Each algorithm yields a different result or different time complexity (the amount of time taken by an algorithm to run).

Discussion

In this project, we implement four different types of search algorithms into the maze to find the most optimal search algorithm for pathfinding. And they are depth first search, breadth-first search, Greedy or Dijkstra search, and lastly, A* search.

1. Necessary Libraries

These search algorithms are implemented in two different modules for visualization. The search algorithm that uses weighted nodes is implemented in pygame whereas the unweighted node can be visualized by using python turtle.

It should be noted that python turtle is a pre-installed library in python. This library allows users to create shapes or pictures on the virtual canvas. The pen that is used to draw on the screen is called the 'turtle.' Whereas, Pygame is a module that is used to create video games. It is easier to assign values to the nodes than using python turtle.

2. Basic Information and Algorithm of 4 Searches

The results show that in terms of weighted node, A* search is more preferable to Dijkstra search for pathfinding in a Depth-First maze. Whereas in unweighted nodes, the Breadth-First Search is more optimal and efficient than Depth-First Search.

These 2 classifications of search algorithms have a major difference. The unweighted node doesn't need knowledge in terms of the searching process, they have no additional knowledge to reach the goal. Which can consume more time and high cost. Whereas in weighted nodes the knowledge to reach the goal node can be obtained by using a function to calculate the distance from the start or current node to the goal node.

In regards to unweighted nodes, Breadth-First Search and Depth-First search falls into this category. Yet, they are different. The algorithm of Depth First search is to explore as far as possible along each branch before backtracking and explore other branches which can be used for generating mazes but most likely not optimal nor efficient in terms of pathfinding. The cost

can be very high to reach the goal in its deepest node. Whereas Breadth-First Search searches used for traversing graphs or trees. The standard BFS algorithm traverses each vertex of the graph into two parts: the first is visited and the second is not visited. So, the purpose of the algorithm is to visit all the vertices while avoiding cycles. Breadth-First Search can be very efficient and low cost if the goal/destination is close to the start node.

In regards to weighted nodes, A* Search and Dijkstra fall into this category. Both of them used the heuristic function to obtain information to estimate how close a node is to the goal node. However, both of these search algorithms aren't the same. In Dijkstra, it explores the path by exploring the closeness node to the goal estimated by h(x). This algorithm doesn't explore all possible paths, but the lower cost path. This could be very useful for multiple destination pathfinding, but not in our case. While in A* search, uses both forward cost (estimate of the distance of the current node from the goal node) and backward cost (the cumulative cost of a node from the start node). In short, $f(x) = h(x) + g(x)$.

3. Program Analysis

Therefore, the algorithm of each search is implemented to the program accordingly. Please see the codes along with this report for better understanding.

A. Depth First Search

In the case of Depth First Search pathfinding, the program will begin the searching from the root node and go down to the leaf of a branch at a time looking for a particular key. If the key is not found, the searching retraces its steps back to the point from where the other branch was left unexplored and the same procedure is repeated for that branch.

Depth First Search or DFS follows 3 steps: 1) Visit a node 'S' 2) Mark 'S' as Valid, and 3) Recursively visit every unvisited node attached to 'S'. Since DFS is of recursive nature, this can be implemented using stacks.

To implement DPS in Python, you must first Push the source node to the stack. Next, maintain a data structure to mark if a node is visited or not. Mark source node S as visited (Visited[S] = True), while the stack is not empty the program will keep repeating the following steps:

1) Pop node, example, A from the stack
2) If visited[A] is True then Popnode. Else, continue
3) Check if A is the node that we need. If yes, break DFS. Else, push all the adjacent nodes of A which are not visited into the stack.

In this way, we will traverse a path until it is exhausted. Meaning, either there are no adjacent nodes or all the adjacent nodes are already visited.

From the project, we import the turtle library then determine the window's size, title, and background color (which is black). Then we use 5 classes to construct a maze including:

White turtle to stamp out the maze, green turtle to show the visited cells, blue turtle to show the frontier cells, red turtle to represent the start position, and lastly, yellow turtle to represent the end position and the solution path. Each class can be split into 5 parts

1) class Maze(turtle.Turtle): def __init__(self): turtle.Turtle.__init__(self)

To define a maze class

2) self.shape("shape name") to define the turtle shape
3) self.color("color") to define the color of the turtle
4) self.penup() to left up the pen so it does not leave a trail
5) self.speed(0) to determine the animation speed

After this, we type in the grid and make a class to construct the maze based on the grid we typed and made the program solve the maze. Setup_maze(grid): is defined as a function called setup_maze. Then use global start_x, start_y, end_x, end_y to set up global variables for start and end locations. Then we iterate through each character in the line, assign the variable character to the y and x positions of the grid. Start moving X and Y location of the screen starting at (-588,288)

If a character contains a '+', move the pen to the x and y location then stamp a copy of the white turtle on the screen, then add a cell to the walls list. If no character is found, add to the path list. If the cell contains an 'e', move the pen to the x and y location then stamp a copy of the yellow turtle on the screen. Assign end location variables to end_x and end_y, add a cell to the path list.

If the cell contains a 's', assign start location variables to start_x and start_y and send the red turtle to the start position. Add the x and y position to the frontier list. Add x and y to the solution dictionary. Loop until the frontier list is empty and change the value to decrease animation's speed; the current cell equals x and y positions.

After this, the program will check left, down, right, and up. After checking and moving forward, it will create the solution path when current cells equal start cells.

B. Breadth-First Search

To implement a breadth-first search in Python, we need a queue data structure. We can conveniently use the deque (double-ended queue) class found in the collections module.

For BFS and DFS, we use the same structure in terms of importing, setting up the screen, class creation, and path of the maze. First, we import turtle library which is a pre-installed Python library that enables users to create pictures and shapes by providing them with a virtual canvas. Then, define the turtle screen and set up the title, dimensions of the working window, and also the background color (For example, wn.bgcolor("black") where wn is defined as the turtle screen and bgcolor stands for background color). After importing the library and setting up the turtle screen we create the class a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state. For the project we do the maze pathfinding so, we set up grid lines in the background by using ASCII characters to assign paths of the maze.

Secondly, in this part of the setting up and constructing the maze, we define a function called setup_maze and set up the global variables for start and end location. The global statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. Then, we use the for statement to iterate over the elements of a sequence (such as a string, tuple, or list) or other iterable objects. The statement for y in range(len(grid)), we used for reading in the grid line by line and the statement for x in range(len(grid)), to read each cell in the line. We assign the variable the x and y location on the grid and also assign the x and y location moving on the screen, for the project we assign x and y to move to the location on the screen, assign x = -588 + (x* 24), starting at point -588 and y = 288 - (y*24), starting at point 288. Next, we use the if statement to assign how the maze goes to (For example, we assign if character == "+": maze.goto(screen_x, screen_y), which means if the character contain plus then move pen to the x and y location then stamp a copy of the turtle on the screen which is white square. If the character is "empty" we will add it to a path list. If the character is "e" for the ending point, stamped by blue square and lastly, if the character is "s", assign start locations variables to start_x and start_y. So, this part is used to determine the

direction of the maze for each character and assign start and end locations variables. After setting up all the constructs of the maze path we do the end program by using def endProgarm(): .

Finally, the part of the breadth-first search, unfortunately, turtle graphics do not use a nice convenient number because we use a grid system. So, we use the x parameter and y parameter instead. For the searching part we use the while loop, which is used when a condition needs to be checked each iteration or to repeat a block of code forever, and then assign that exit while loop when frontier queue equals zero then pop the next entry in the frontier queue and assign to x and y location, we use x, y = frontier.poplest() to assign this. After that we check the cell on the left that is in the path or the wall and also right, up and down (For example if(x-24, y) in path and (x-24, y) not in visited) which means to check cell goes back to the left on the x-axis is in the path or the wall.

Lastly, a backtracking routine solves the problem by "reverse" searching every possible path, to find out if the answer is correct. If not, then go back to the same spot and look for new answers.

C. A* Search

In the case of A* pathfinding program that uses pygame, these are the strategy that is applied into the program to implement the A* algorithm for pathfinding.

Similar to the previous algorithm, the colors will be assigned first, in this case in the class Spot, the RGB values are stored in the colors name variable. And then assign the colors to the nodes. (For example, def make_path(self): self.color = PURPLE).

Moreover, unlike python turtle, pygame doesn't have a virtual canvas and we need to create that canvas by ourselves. Using the method make_grid with rows and width as its parameters. The purpose of this method is to set the number of rows and columns as well as the width of the grid to be. Performing an integer division between width and rows will give us what the gap should be between each of these rows, in other words, what the width of each cube should be.

Furthermore, the loop in this method creates a two-dimensional list where a bunch of different rows is in their list and they will have spot objects inside of them. And to draw the grid, a pygame function will be used. By using pygame.draw.line to draw the grid line, both horizontal lines and vertical lines under the for loop (draw_grid method). The main draw function will draw everything, the whole canvas. And uses pygame.display.update to update the window of what the players have drawn.

Of course, there should be a function to determine the mouse position so the program will be able to figure out which spots/cubes to change colors when players click the mouse. The function (def get_clicked_pos(pos, rows, width):) will determine the mouse position and translate it into coordinates of the grid. To determine the position of mouse clicks on the grid, by taking the position is in the x and y then divide it by the width of each of the cubes. This will give us the position of the cube we clicked on in rows and columns.

In this A* pathfinding, manhattan distance is used to find the distance between two nodes, the start, and the goal node. The Manhattan distance is the sum of the absolute differences between the two vectors. (In a method, def h(p1, p2):)

Now, in the main method (def main), the main algorithm of this program will start by defining the number of rows that the players want, and pass the make_grid method to generate the grid and give a 2D array/list of spots. Next, the start and end position will be defined, in this case, it's None since the players will decide by themselves where the start and end node will be when the program runs. And while run is true, at the beginning of each of the while loop, it will loop through all the events that have happened and check what they are. This is what the loop will check, if the players quit (press X at the top right-hand corner) the game will stop running and exit the window.

Next, the program will check if the player clicks the mouse or not. If players **click mouse button 0 (left button)** according to the mouse position, the program will draw and mark that spot as either the start node, end node, or the barrier and make sure that these 3 nodes type will not override each other. Otherwise, if the **player clicks mouse button 2 (right button)** it will

reset the spot to be white (clear). And if the player presses the **keyboard's spacebar**, the algorithm will start to run.

However, to run the program, the program should know the node's neighbors and recognize that the barrier (obstacle) is not the node's neighbor. In def update_neighbors in class Spot will add neighboring nodes to that current node. Inside this method, the program will check if there are nodes nearby to append (up, down, left, right). Now, the main algorithm can recall the function update_neighbors for the program to find the possible path from that current node. And lastly called the algorithm and passed the draw function as an actual argument to another function.

Now, the def algorithm can easily call the draw function. This is where PriorityQueue will play its role. It will add the start node with its f score into the open_set as well as count to keep track of when we insert items into the queue and can break ties if there are two things inside of the queue that have the same f score, the program will go based on which was inserted first. Then we have the came_from dictionary for keeping track of which node we came from. Next, set up the g and f scores. The g score of the start node is set as 0 whereas the f score, the start node is according to the heuristic distance from start to end node. In open_set_hash use the set to check if there's something in the queue or not. The algorithm runs until the open set is empty.

When the open set is empty that means the program has considered every single possible node that has been explored, and if the path is not found, the path doesn't exist (The player can close the window anytime too since there's a pygame.quit inside this loop). The node that just popped out of the priority queue, synchronizes with the open set hash to make sure there are no duplicates or anything out of sync. And if the node that just pulled out is the end node, that is the shortest path and the path will be constructed and displayed on the window.

Note that in this program assume all the edges are 1. In the for neighbor loop, it helps to find the better path. Then the program will add that node (as well as its row and column position) if it's not in the open set.

And finally, the path will be constructed considering all the neighbor nodes and finding the shortest path to reach the end node.

D. Dijkstra Search

While in Dijkstra pathfinding program, It's kidda similar to the A* algorithm. We also use pygame but used it in Dijkstra Search for pathfinding.

So, the coding in this search we try to do something different. We didn't assign the color like the previous algorithm, whereas we decided to spot the color by using RGB color to assign the node. (Ex. spot.show(win, (128, 0, 128))

In addition, We set up about how many boxes we are going to have in one line and just set up everything to prepare for the data.And to try doing a new activity for a time, we decided to assign the beginning and the endpoint in this algorithm. (Ex. start = grid[0][0], end = grid[20][10])

The Spot class as we told that we are using pygame in this algorithm, so we have to create that canvas by ourselves like A*search. As we use pygame to run this algorithm, we are going to use lots of pygame functions in this part. For example, we use pygame.draw.rect/pygame.draw.circle to draw rectangles and for the add neighbour function, we use this to make the grid. Then, we just create a clicked function to draw the main in our main function.

For the main function (def_main), we have the function to set the mouse position as A* algorithm also did, to be able to show the maze that the user wanted to make. Since in this program we decided to assign the beginning and ending point already, we do not need to create the function to figure out when we have to change the color. Next, we use clickWall(pygame.mouse.get_pos(), event.button == 1) to draw the maze and clickWall(pygame.mouse.get_pos(), event.buttons[0]) to erase. Then, we set the program that when we are pressing **SPACEBAR** startflag=True by using event.key == pygame.K_SPACE. So, this will link to the next code which program is going to run. We are using while to find the

path as in this project each rectangle's size is all the same so no need to create complex functions in this program. So, we just find the path and we just want to add some more ideas by setting it print(“Done”) on the program, if the program can find the path, whereas if not we decided to add this code messagebox.showinfo("No Solution", "There was no solution") to represent that the program cannot find the path.

Lastly, we just paint the color for each part to make it easy for the user to easily understand how the program finds the path and all the lines to partition each rectangle and we have pygame.display.flip() to update the full surface of the screen.

Nevertheless, The objective of the Dijkstra algorithm is to find the shortest path between any 2 vertices in the graph. In fact, the Dijkstra algorithm will find the shortest from the given starting vertex to the other vertex in the graph, which as you can see in our objective in this project the size of each rectangle is the same so, we think that it might not suitable to use Dijkstra algorithm in this project because we do not have to calculate the distance between each rectangle and in this algorithm just explore all possible paths which we actually might not have to identify all of those paths.

Conclusion

The purpose of this project was to identify the most optimal or efficient search algorithms for pathfinding, in this case, a simple maze. In this project, 2 different modules, pygame, and python turtle are used to help visualize how each search algorithm explores its most efficient or most optimal route from the start node to the goal node.

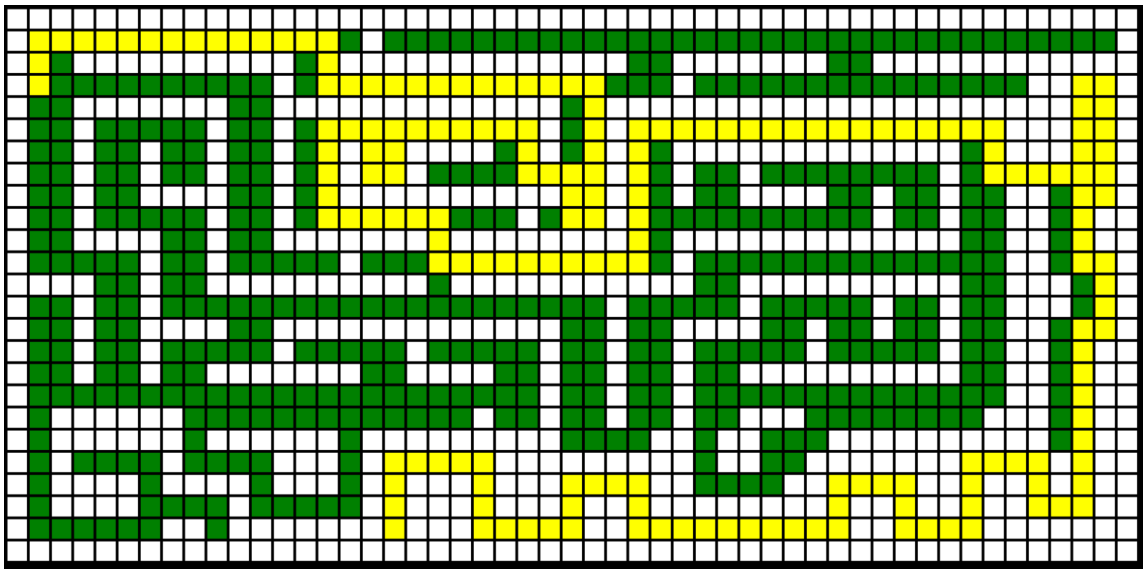
Based on the results and algorithm of the program it can be concluded that A* will search its most efficient path by using the heuristic function to find the solution path with the least cost whereas Dijkstra will explore all possible paths before finding the most optimal route. This makes A* search the solution path faster than the Dijkstra.

On the other hand, Depth – First Search uses stack structure. It will explore the deepest node in the branch before backtracking to other possible branches. Unlike Breadth-First Search, it uses a queue structure. Breadth-First Search searches through each tier level. When the remaining sum is zero and the node is a leaf, then we push the path to the result. It can be very efficient if the goal node is closest to the start node, the BFS algorithm can easily create the shortest path and there is nothing like a useless path in BFS since it searches level by level. And this is what makes the Breadth-First Search more optimal and efficient than the Depth-First Search.

Appendix

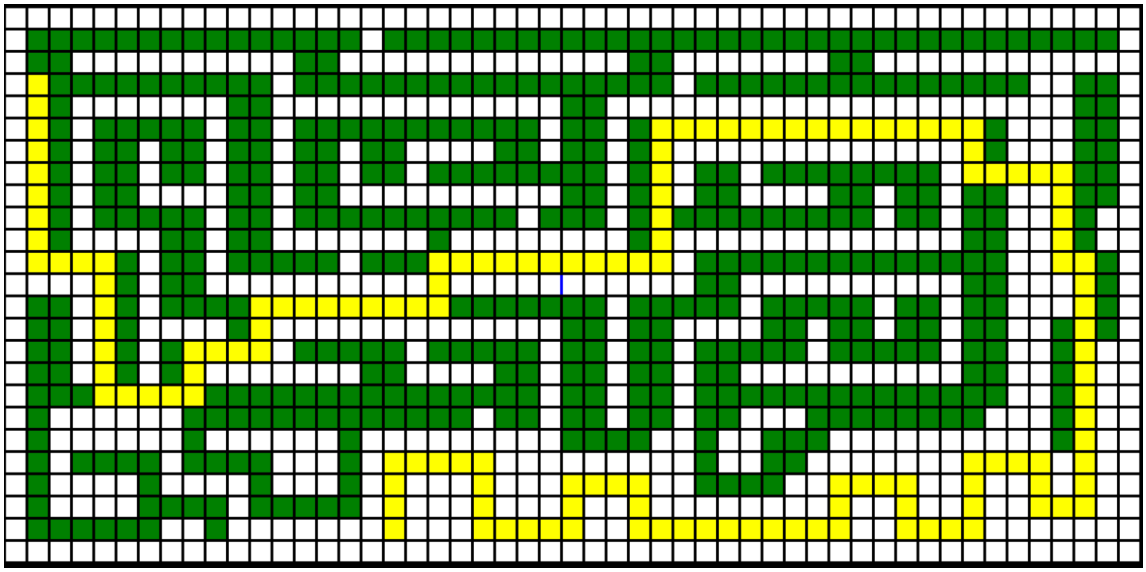
Appendix A

Depth-First Search, compute time: 40 seconds (including generating the maze)



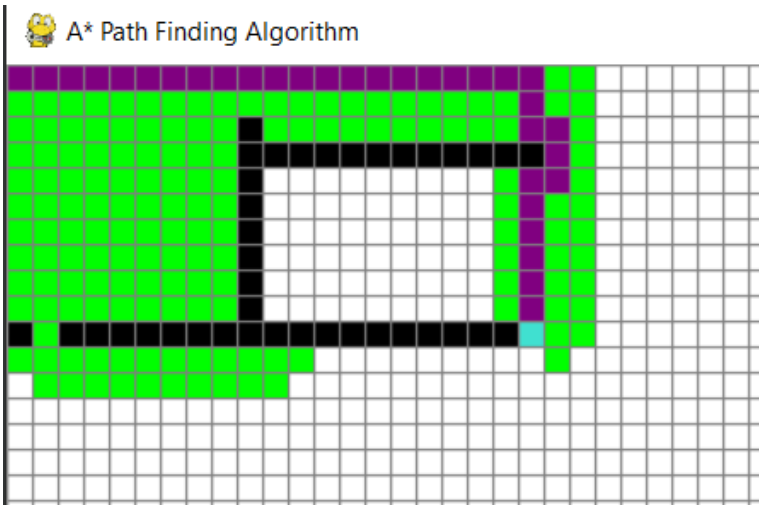
Appendix B

Breadth-First Search, compute time: 27 seconds (including generating the maze)



Appendix C

A* Search Algorithm, compute time: 2 seconds



Appendix D

Dijkstra Search Algorithm, compute time: 2 seconds (with more nodes explored)