# CSC111 Project: Utilizing Computer Vision and Graphs to Solve Arbitrary Mazes

Ifaz Alam, Tony He, Austin Blackman

Tuesday, March 16, 2021

## Problem Description and Research Question

Within the second course based on the Foundations Of Computer Science (CSC111), students learned how to use graphs to represent networks of connected data. Specifically, a graph contains data, known as vertices, and connections between the data known as edges. Popular examples utilizing graphs include visualizing connections between cities, flight paths, and which actors starred in movies together.

**For our final project within CSC111, our group will investigate how we can relate graphs to mazes. Specifically, we wish to be able to find an image of a maze on the internet or draw a maze on paper, input into our program, and find the shortest possible path required to complete the maze. Additionally, we want to be able to visualize each step the program takes in order to complete the maze. This step is vital as it will allow users to see how each path finding algorithm works, and thus will be able to easily compare them**.

In order for the reader to understand this problem and our solution to it, some knowledge of computer vision will be necessary. This is due to the fact that we will need to be able to take an arbitrary image from the internet or the users hand drawn image, and turn that image into data that we can analyze and operate upon. Additionally, the reader will also need to understand loops. This is because we will need to find and implement numerous path finding algorithms in order to find the shortest possible path to solve a maze. We will also need to compare the algorithms to determine which one is fastest, and fits our needs the best. Lastly and most importantly, the reader will also need to understand graphs. This goes hand in hand with pathfinding, as we will utilize how edges and vertices are related in order to implement and investigate the different pathfinding algorithms.

## Computational Overview

Modules/Libraries:

- opencv-python and opencv-contrib-python. The former module is used for thresholding and preprocessing the image, and the latter module is used because it contains an implementation of the Zhang-Suen thinning algorithm written in C++, which is essential since it is a costly algorithm (if image resolution is too large) and would take too much time if written in Python.
- numpy for array operations
- numba's nopython mode complies specified python code into machine code the first time it is run. This is very important as methods like `get_neighbours()` will be called thousands to millions of times and so running the functions from complied code is essential to keep up with real time visualization.
- pygame for real time visualization

The program first takes the specified maze image and (if the maze is rectangular) crops the maze such that the outer white space is removed. This is done by finding the contours of the image, and assuming that the two longest lines will be borders of the maze (usually this is the case for rectangular mazes, so the assumption is okay). The program does work with any shape maze, however, the white space around the maze may lead to weird paths. Otsu's binarization is then applied to isolate the maze from the image, and then the maze is thinned to 1 pixel using

Zhang-Suen algorithm to preserve connectivity. The perseverance of maze connectivity is integral to the program since any "holes" in the maze will alter the path the program finds. The resultant thinned maze is then stored into a 2d array which represents a graph where each pixel of the maze is a vertex, and the edges are the valid path nodes from of the neighboring 8 pixels.

The MatrixGraph class stores this array, and has various methods that are useful for graphs. For instance, getting the neighbors of a vertex, finding the euclidean distance between two vertices within the graph, and calculating the closest point of a path in relation to a specific point.

The PathfindingAlgorithms class within the algorithms.py module has various path finding algorithms as methods:

1. Breadth First Search

   This version of breadth first search is implemented iteratively. The algorithm starts at the designated start point, adds that vertex to a queue and marks it as visited, along with all its neighbors and then loops through the queue until it is empty, or the target is found. The loop checks if the first vertex in the queue is the target vertex, if it is not, then the vertex is removed from the queue and the vertex's neighbors are added to the queue if they have not been previously visited. Additionally, at each iteration of the loop, the path searched is drawn in red. The function also keeps track of the path that is searched, so the final path can be drawn in green and returned. The breadth first search algorithm searches each vertex in a first in first out order according to the queue. This allows the path to quickly spread across the entire graph.

2. Depth First Search

   This version of depth first search is implemented iteratively. The algorithm starts at the designated start point, and adds that point to a stack. The function then loops until the stack is empty, or the target is found. At each iteration of the loop, the top vertex of the stack is checked to see if it is the target vertex, if it is not, the vertex is popped and it's neighbors are pushed into the stack if they had not been previously discovered. Additionally, at each iteration of the loop the path searched is drawn in red. The function also keeps track of the path that is searched, so the final path can be drawn in green and is returned. The depth first search algorithm searches each vertex in a last in first out order according to the stack. This allows for the algorithm to cover a large distance but skips over side paths.

3. A*

   An unweighted implementation of the A* search using the Euclidean Distance heuristic is implemented. The reason we opted for euclidean distance over Manhattan distance is because the method used to crop the maze is limited to rectangular mazes; everything else works with any kind of maze. Thus, diagonal lines in mazes are possible and so Manhattan distance is less accurate in these situations.

All visualization within the program is done via Pygame. The following modules all utilize Pygame in order to convey data to the user, and allow for user input.

- button.py: This module contains the Button and ToggleButton class that allows the user select the start and stop points of the maze, along with reset the inputs.
- clock.py: This module contains the Timer class which allows the user to inspect how long the specified algorithm has been running for.
- text_box.py: This module contains the TextBox class which allows the user to manually input the filename of a maze via the keyboard.
- drop_down.py: This module contains the DropDown class which allows the user to select which pathfinding algorithm and maze they want to see the program run with.
- algorithms.py: This module contains the PathfindingAlgorithms class which as previously mentioned stores the pathfinding algorithms our program utilzes, however it also contains methods such as _draw_loop_iterations which allows the user to note how many nodes have been searched at any given point.

# Instructions for running the program

Required packages and their pip install command.

1. Numba

   pip install numba

2. OpenCV

   pip install opencv-python

   pip install opencv-contrib-python

3. Numpy

   pip install numpy

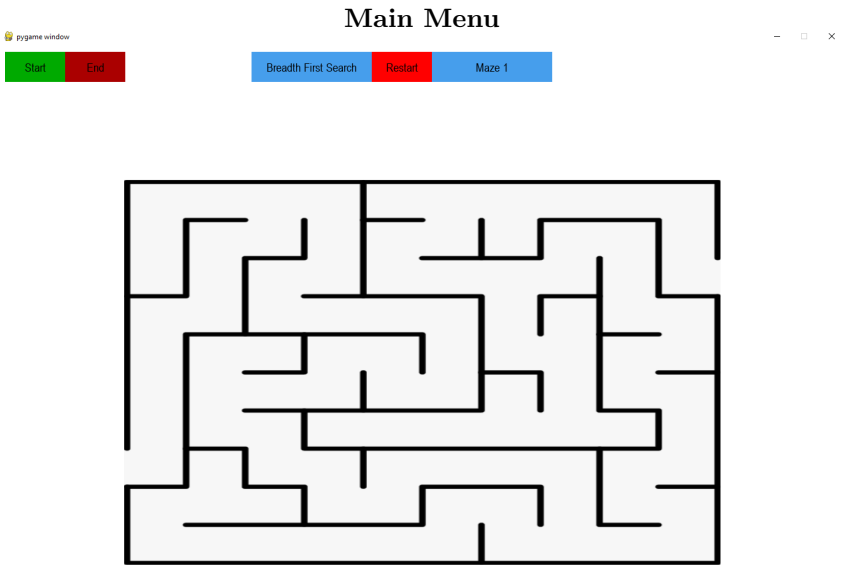
4. Pygame

   pip install pygame

5. Python_ta

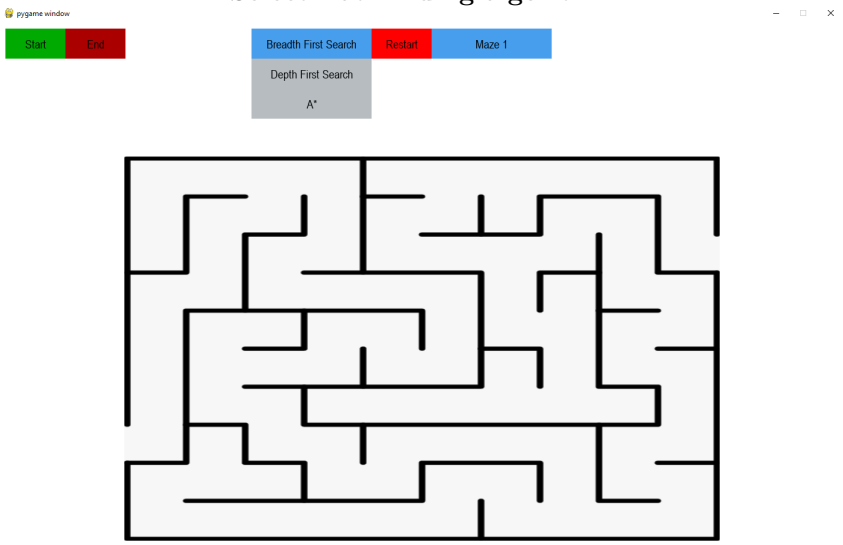
Setup: Place all the .py files in one folder, create a folder called mazes and place it within the same file as the .py files. Within the mazes folder, place the maze images.

To run the program, run the main method, main.py. The user will be greeted by the menu of the program with the first maze loaded.

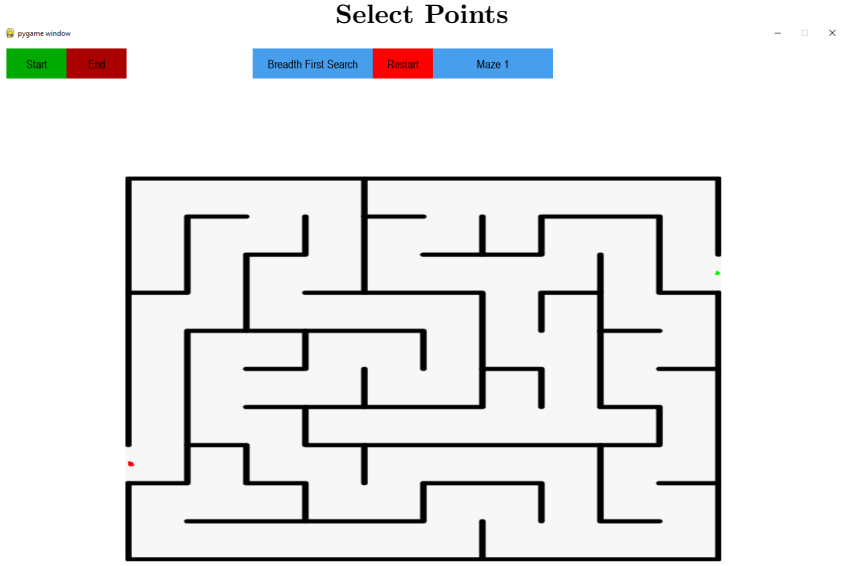



The user can then select their desired pathfinding algorithm (by clicking on the drop down menu) and start and stop points by clicking the start / stop button, then the point on the maze.

**Select Pathfinding algorithm**

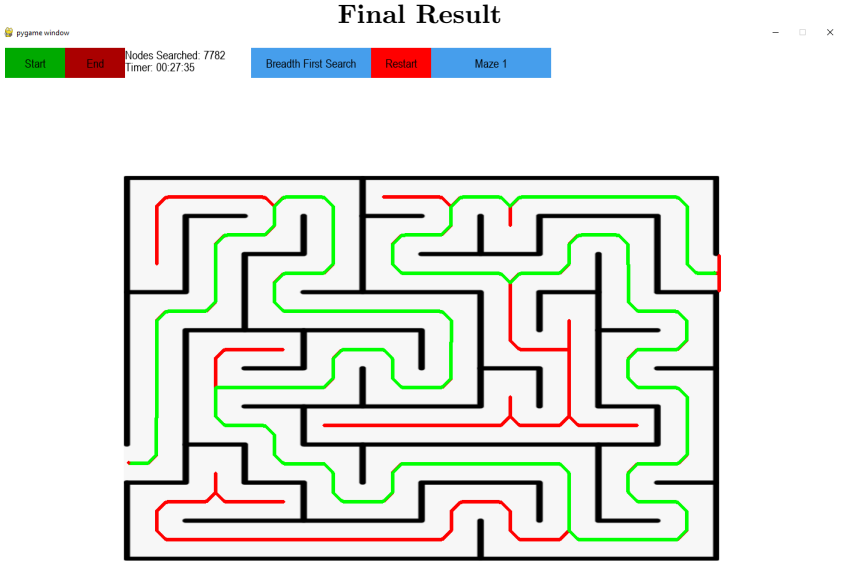



**Select Points**




Once both points are selected (and valid, if they are not, a message warning the user will be printed to the python console) then the visualization will start and the path the algorithm traces will be drawn in red. Once the algorithm finds the stop point, the final path will be traced in green. To run the program again, or with a different maze and or pathfinding algorithm, click the restart button and repeat the previous steps.

**Final Result**



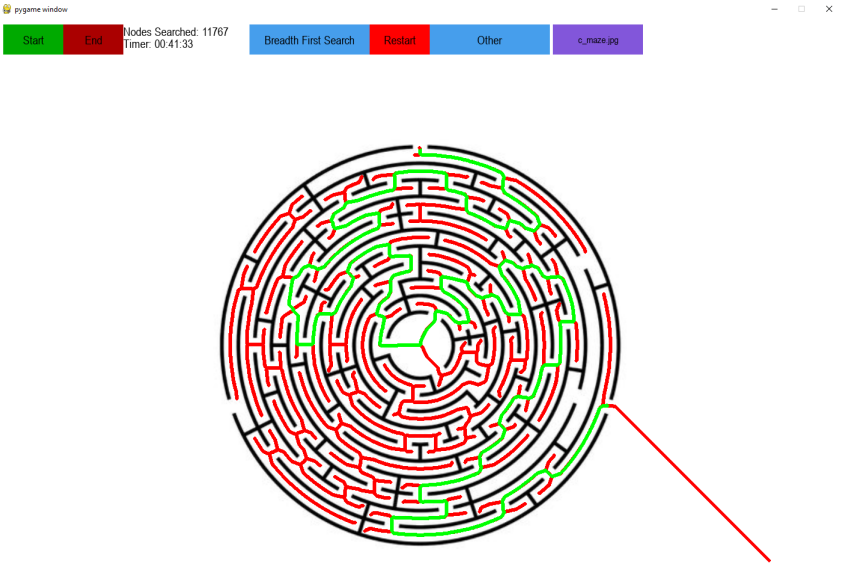

## Using Custom Mazes

The program also supports using the users own images. To do so, find or create an image of a maze that has a white background and black walls. The image must be of an appropriate resolution, and rectangular mazes work best (if using a non-rectangular maze, please view the subsection, 'Using Circular / Non-Rectangular Mazes'). Place the image into the /mazes folder. Relaunch the program, and click the drop down menu for maze selection. Pick 'other', and then within the text box that appears, type the name of the file, including the extension and press the ENTER key. Once the maze loads, you may follow the steps outlined above to see your custom maze visualized!

## Using Circular / Non-Rectangular Mazes

The program supports circular / non-rectangular mazes, although the cropping algorithm used does not support circular / non-rectangular shapes as mentioned in the computational overview. Therefore, when adding circular / non-rectangular mazes into the /mazes folder, please make sure the name of the file begins with 'c_', for example 'c_my_maze.png'. This prefix tells the program to skip the cropping step.

**Circular Maze Example**



# Changes between proposal and final submission

Between the final submission and the proposal, the only change our group implemented was to no longer include recursive pathfinding algorithms, which is discussed further in the **limitations** subsection within the **discussion** section

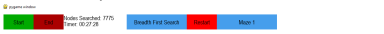
# Discussion

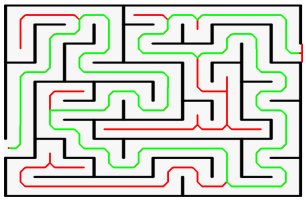
## Results

From our exploration into how our program would work, we quickly found that the graphs and pathfinding algorithms go hand-in-hand as pathfinding algorithms were actually intended to be used on graphs.

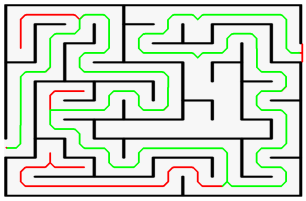
As for the comparison between the various path finding algorithms we found that with simple mazes, such as maze 1 (mazes/maze.png) depth first search is the clear winner when the start and end of the maze are on opposite sides. As you can see from the diagrams, the middle section is completely skipped with depth first search, allowing for roughly 1100 nodes to be skipped in comparison to BFS and A*, which allows for about 6-7 seconds to be saved.

**Maze 1 - BFS (7775 Nodes Searched, 00:27:28)**

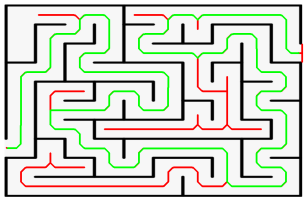




**Maze 1 - DFS (6579 Nodes Searched, 00:20:46)**

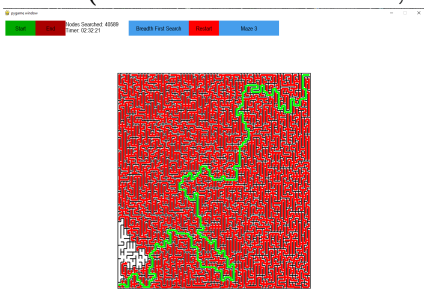


**Maze 1 - A* (7602 Nodes Searched, 00:26:85)**



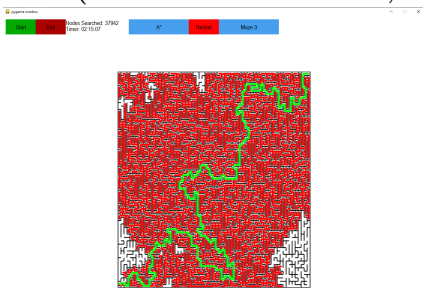
For more complex mazes such as maze 3 (mazes/maze3.png), when the start and end points are selected at opposite sides of the maze, depth first search is still the clear winner, even more so than the previous simple maze. Depth first search was able to skip roughly 15800 - 18000 nodes in comparison to BFS and A*, which saved roughly a minute and fifteen seconds in comparison for time.
However, now that the graph is more complex, A* begins to perform much better than BFS, as its distance calculating heuristic is able to be utilized. Notice in the diagrams below that A* was able to find the end of the maze on the bottom left without searching the bottom right. Compared to BFS, almost the entire maze is searched.

**Maze 3 - BFS (40589 Nodes Searched, 02:32:21)**



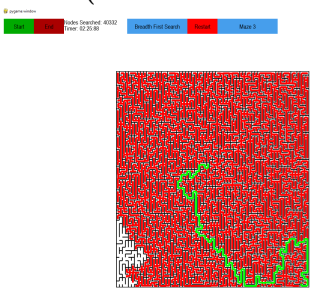
**Maze 3 - DFS (22187 Nodes Searched, 01:19:52)**

**Maze 3 - A* (37942 Nodes Searched, 02:15:07)**

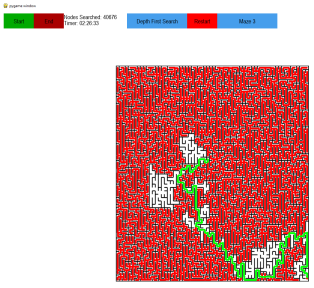


Interestingly, we know that A* should be out performing both other algorithms since it takes into consideration more factors when choosing a path to search. However, since we are searching a maze, we might need to travel in a direction that is opposite to the target vertex, thus A* does not perform to the best of its capabilities. However, we can devise a test to show that A* performs better than DFS and BFS in certain situations, such as when the algorithms start in the middle of the maze, and attempt to find the target on an edge. In this case we can see that A* is marginally better than DFS and BFS, searching roughly 6000 nodes fewer, and saving roughly 40 seconds.

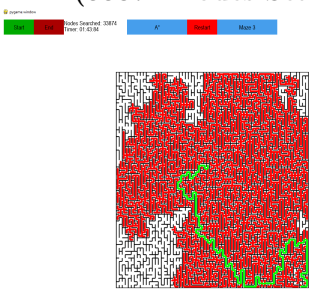
**Maze 3 - BFS (40332 Nodes Searched, 02:25:88)**



**Maze 3 - DFS (40676 Nodes Searched, 02:26:33)**



**Maze 3 - A* (33874 Nodes Searched, 01:43:84)**



To conclude, we found that mazes and graphs are extremely well related, as we can convert the maze into a graph, and use pathfinding algorithms to find a path through the graph. As for which pathfinding algorithm we found to work best on mazes, we found this to be inconclusive as the nodes searched by each algorithm heavily depends on the maze structure, and where the start and end points are located, as mentioned above.

## Limitations

When implementing the pathfinding algorithms into our program, we started with recursive algorithms because that would be most fitting for CSC111 since the course focused on numerous recursive algorithms. However, we found that the recursive algorithms would throw a RecursionError. The reason why this happens is because our graph is extremely large, as there are thousands of pixels that need to be searched, and thus the maximum recursion depth is exceeded, much like what was found in Tutorial 7 when trying to recurse on a large graph of actors. Therefore, we had to change our pathfinding algorithms to use an iterative approach instead of a recursive approach in order to avoid errors.

Thanks to the approach taken to generate a maze, any kind of maze will work. However, there is sometimes a problem with depth first search. In diagonal sections, Zhang's method sometimes thins the path into a staircase-like pattern, as opposed to a true diagonal line where only the vertexes of each pixel touch. This causes a problem

because diagonal nodes are considered neighbours and so the staircase pattern results in 2 possible traversal paths when there only should be one. This is only an issue for depth first search as it will search a path until a dead end and possibly miss the target.

Due to the way our visualization is implemented, the program updates 1 searched node per frame even when there are multiple exploration paths. As a result, both breadth first search and A* look like they are lagging while multiple branches of the maze are being searched at once. Instead of updating 1 node in every branch of the search at once, it must update them one at a time. This makes it look slower, however the same number of nodes are still being searched in a given period of time compared to depth first search.

# References


Anastasia Murzova Sakshi Seth, Murzova, A., & Seth, S. (2020, August 26). Otsu's thresholding Technique: Learn OpenCV. Retrieved March 09, 2021, from https://learnopencv.com/otsu-thresholding-with-opencv/

Clark, R. (n.d.). Maze Generation with Eller's Algorthm. Retrieved April 16, 2021, from https://clarkcoding.com/eller.html

Garbutt, B. (n.d.). Maze PNG page - Transparent Mazes, Png download - kindpng. Retrieved April 16, 2021, from https://www.kindpng.com/imgv/iiiTmxb$_maze-png-page-transparent-mazes-png-download/$

Jessica Boddy August 19, 2. (2021, March 22). This maze has three SOLUTIONS. Can you solve them all? Retrieved April 16, 2021, from https://www.popsci.com/story/science/head-trip-always-another-way-in/

Numba documentation. (n.d.). Retrieved March 09, 2021, from https://numba.pydata.org/numba-doc/latest/index.html

OpenCV modules. (n.d.). Retrieved March 09, 2021, from https://docs.opencv.org/master/

Sarmdy. (n.d.). Illustration of maze, labrinth. isolated on white background. medium... Retrieved April 16, 2021, from https://www.istockphoto.com/vector/illustration-of-maze-labrinth-isolated-on-white-background-medium-difficulty-gm881659914-245444779?irgwc=1amp;cid=ISamp;utm_medium=affiliateamp;utm_source=TinEyeamp;clickid=wkqwoyR7lxyLTh

Zhang-Suen thinning algorithm. (n.d.). Retrieved March 09, 2021, from https://rosettacode.org/wiki/Zhang-Suen_thinning_algorithm