Ofer Moses and Igal Edreaii

**Maze Solver**

No Maze left Unsolved!

[mosesofe@post.bgu.ac.il](mailto:mosesofe@post.bgu.ac.il) [igale@post.bgu.ac.il](mailto:igale@post.bgu.ac.il)

**Introduction**

Life is full of walls, and sometimes we have to figure out the fastest way around them.

For this propose , we introduce the maze solver!

First, a little about mazes:

They can be made of hedges, corn, wood or mirrors. They can be spiritually calming or visually stimulating, and they can incite feelings of panic, excitement or serenity. Mazes have been a part of human culture for thousands of years—but what compels us to wander through winding pathways in order to find a single, hidden exit?

Mazes have an [ancient history](https://www.davisfarmland.com/megamaze_2011/files/history.pdf) spanning thousands of years— The first recorded labyrinth (Maze with a single winding path) comes from Egypt in the 5th century B.C

During the Middle Ages, labyrinths evolved from spiritual journeys to amusing pastimes. As kings and queens built up elaborate gardens, they would often include some sort of hedge maze as a diversion for themselves and guests.

But what is the definition of a maze?

A **maze** is a path or collection of paths, typically from an entrance to a goal.

**Maze solving** is the act of finding a route through the maze from the start to finish. Some maze solving methods are designed to be used inside the maze by a traveler with no prior knowledge of the maze, whereas others are designed to be used by a person or **computer program** that can see the whole maze at once.

The [mathematician](https://en.wikipedia.org/wiki/Mathematician) [Leonhard Euler](https://en.wikipedia.org/wiki/Leonhard_Euler) was one of the first to analyze plane mazes mathematically, and in doing so made the first significant contributions to the branch of mathematics known as [topology](https://en.wikipedia.org/wiki/Topology).

So whether you are out on a spiritual quest or just a maze enthusiast that wants to have some fun like a middle age king, The Maze Solver Program can help you in any of the following objectives:

1. Find a path solving very complex mazes that are hard for us humans to discover, due to various reasons such as: very large mazes, multiple convoluted paths ,dense mazes and constantly needing to perform backtracking when meeting a dead end!
2. Discovering new paths between any 2 points of choice inside the maze
3. Not only finding a path between 2 points inside the maze, but finding the shortest possible way!

For us, the purpose of this project was to provide a fast and easy solution for these objectives, by utilizing the knowledge we gained from this course on Computer Vision skills such as Image Processing, edge and contour detection, along with Computer Science principles of finding the most optimal solution , in our case finding the shortest path through a given maze in the most efficient way possible.

Our solution included creating an application with:

* a user interface, allowing the user to upload an image of a maze of his choice
* allowing a choice of any start and end points for a path
* Finding the shortest path between the input points, and displaying it to the user drawn upon the original image , with the option of saving the solution.

**Approach and Method**

User Interface Side:

In the Main window of the application, we allow the user to upload an image of a maze in the supported formats (jpg, jpeg and png). We also provided a mazes folder with various examples of interesting mazes to solve.

After uploading an image, the user can select with the radio buttons the “Start Point” or “End Point” options, and click on any 2 inner point inside the maze to select them. The coordinates will be displayed in the bottom, and a red and blue circles will mark the chosen spots.

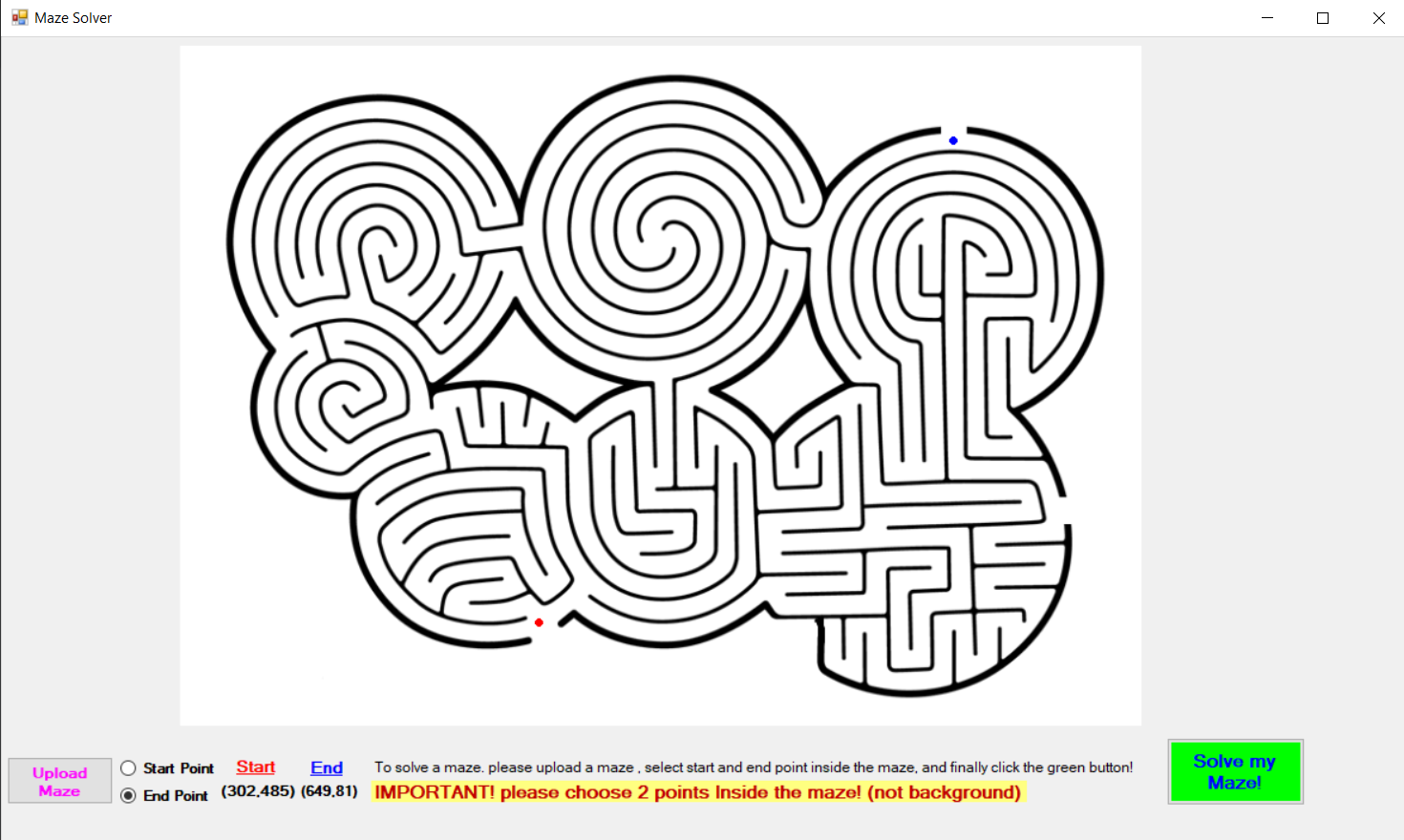
Finally, upon selecting the points, the user can select the Green “Solve my Maze!” button, that will activate the python script that will receive the relevant parameters:

* Start and end points coordinates
* File path of image
* Size of the image displayed in the UI ( image is resized to fit the window properly, therefore the python script will require this size in order to recalculate the point coordinates based on the original size)

After the python script completes , if solving was successful the solution will be saved in the “tmp\_solved” folder, and a window of the default image viewer will pop up displaying the solution to the user, with an option to save in the desired location.

In case of a failure, the python script will send a message to the standard output, and an error message with the relevant Details will be displayed (for example invalid image/start end points).

\* On the first time running the “Solve My Maze!” option, before solving the maze a message will appear asking the user to approve installing the relevant requirements for the python script (OpenCV, NumPy, etc..). this choice is optional , but in case of choosing “NO” the program may fail to due missing this requirements (specified in the requirements.txt file included)

Upon exiting, the program will clean the “tmp\_solved” folder.

Maze Solving Side

First we shall describe in short the main steps of the algorithm:

1. Parse input parameters
2. Read maze image from input path, and recalculate start and end coordinates based on the relations of the UI image size, and the original size.
3. Crop the Background of the image based on the smallest bounding rectangle containing the maze, and again recalculate coordinates accordingly.
4. Get a range of sizes to try resizing the maze to before solving
5. For size in resizes list:
   1. Resize image and calculate new coordinates
   2. Run Edge Detection
   3. For each distance (representing minimal distance from walls):
      1. Attempt solving the maze using a BFS algorithm.
6. If we solved the maze:
   1. Draw the solution on the original image.
   2. Save the image in the “tmp\_solved” folder
   3. Send a message of success back to the UI and exit
7. Else send an appropriate fail message and exit.

Now we will Describe in more detail the steps that are more interesting and complex:

Clarifications: throughout the report and python code, we denote X as the row axis of the image matrix (or Height), and Y as the column axis (or Width).

2. recalculating coordinates after resizing : throughout the program, we have several times where need to calculate the coordinates of the start and end points, after resizing the maze image, we do so according to the following formula:

3. Crop process: (crop\_background method in code)

We used cropping on the maze object in order to remove the background, which helped us achieve 2 goals:

1. We better avoid “illegal” solutions that could happen in some mazes that have openings, where the shortest path found be the BFS could in fact go outside of the maze itself, through the background.
2. We improve runtime due to decreasing the number of pixels, by avoiding to check paths that go through irrelevant background pixels.

In order to crop the image we want to find the external contours of the maze itself, in order to be able to distinct it from the background.

we use several methods from the OpenCV library.

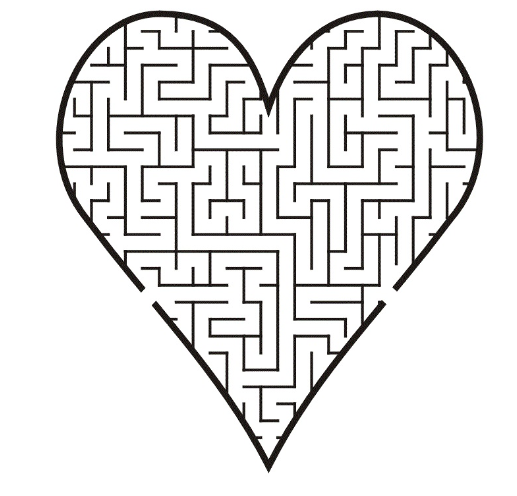
a.Firstly, we use the cv.Threshold method to turn the image to binary (foreground and background).

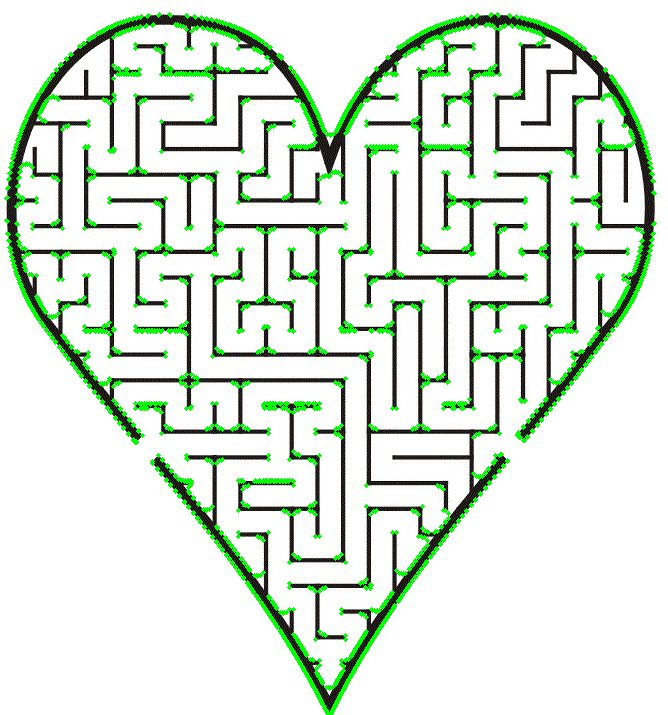
b.Secondly, we use a Morphological Transformation of Closing, which uses Dilation then Erosion in order to “close” small holes found in the foreground object , in order to better identify the contours in the next stage. The dilation slightly “increases” the size of the maze object, making small holes be “filled” or making the pixels between the holes “touch”. Then , Erosion decreases the maze object size back, while keeping these holes “full”.

More Explanation on the Transformations:

Erosion: erodes away the boundaries of foreground object by sliding a kernel through the image (as in 2D convolution). A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero). So what happends is that, all the pixels near boundary will be discarded depending upon the size of kernel. So the thickness or size of the foreground object decreases or simply white region decreases in the image. It is useful for removing small white noises

Dilation: is just opposite of erosion. Here, a pixel element is '1' if at least one pixel under the kernel is '1'. Therefore, the size of foreground object increases

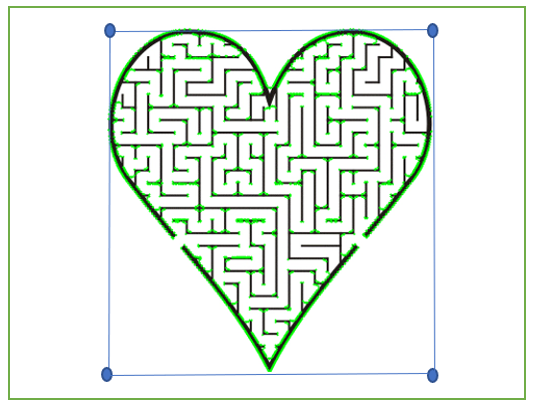
c. we use cv.findContours with the cv.RETR\_EXTERNAL parameter, which gives us an array representing the coordinates of the external contour pixels of the maze object (white pixels).





d. use cv.BoundingRect to receive the coordinates of the “vertices” of the smallest rectangle that contains the whole maze object, then crop the image keeping only this rectangle.

We will denote the upper left vertex as:

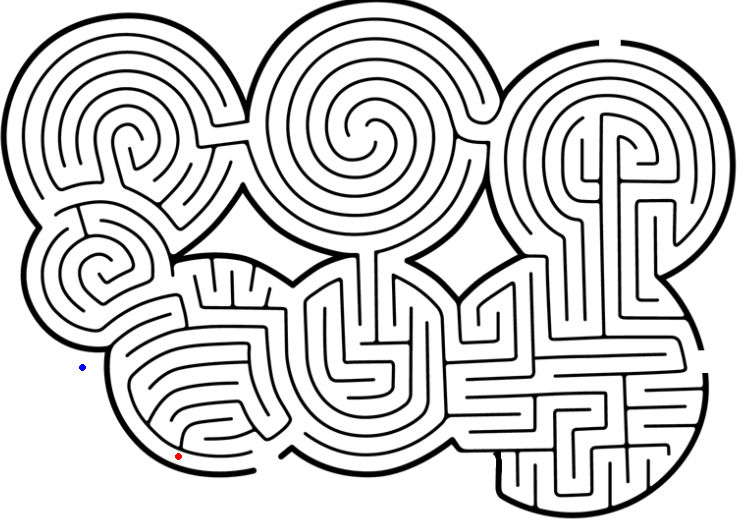


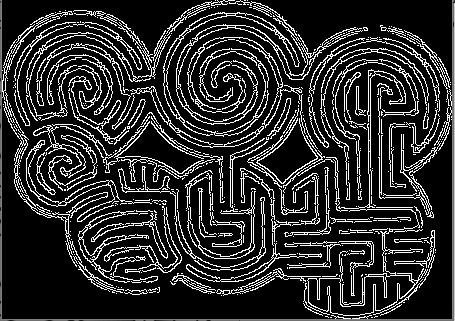
\* Originally , we intended on cropping only the exact maze shape(heart in the example), and to consider all of the background as a wall, in order to better improve performance and avoid solutions “outside the maze”. But , as seen in the example we sometimes get contours with “holes” in the shape frame and inside it , therefore this method often made us ruin the original maze, and we chose the rectangle as a safer option less sensitive to “noise”. In order to better avoid illegal external solutions, we instead added a bounding cross, as we shall explain in the edge section.

e. Finally , we recalculate the coordinates of start and end points after cropping based on the following calculation:

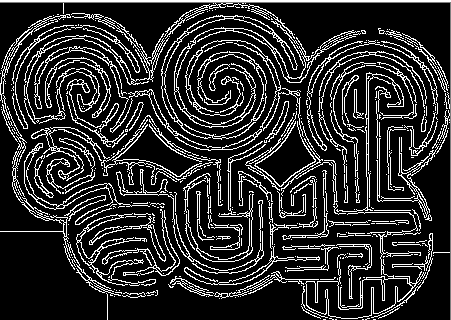
(or in words, we decrease the difference between the crop window and original image)

4.+5a.Resizing range: before solving , we attempt to resize the maze image to a smaller size, or to “compress” the image, while maintain it’s original relations between height and width. We do this in order to greatly decrease the runtime of the BFS algorithm (reducing pixels amount). Because there is a possibility of decreasing the maze too much and failing to solve , we attempt several sizes in increasing orders until success.

5b. Use Canny edge detection in order to find edges of the maze, in our case these would be the “walls” of the maze we will try to avoid when searching for a path. We chose Canny because we knew from previous projects in the course that it would provide good results for our needs.



In addition to the edge map received from Canny, we added a sort of “cross” edge to avoid solutions that go outside the maze, through given openings. we did this in the following way: for a start and end pixels , we looked for the most external edges between ,on both X and Y axis (up and down for the X axis, left and right for Y axis). From each point, we drew a line extending all the way to the end of the image, which blocks any path from outside the maze. For example:



in this example,These lines (left and down cross lines) prevented the illegal solution

5c. we kept a variable d, representing the minimal distance our path can have from any walls, And attempted solving the maze with decreasing d values, until success. The reasons for this choice were:

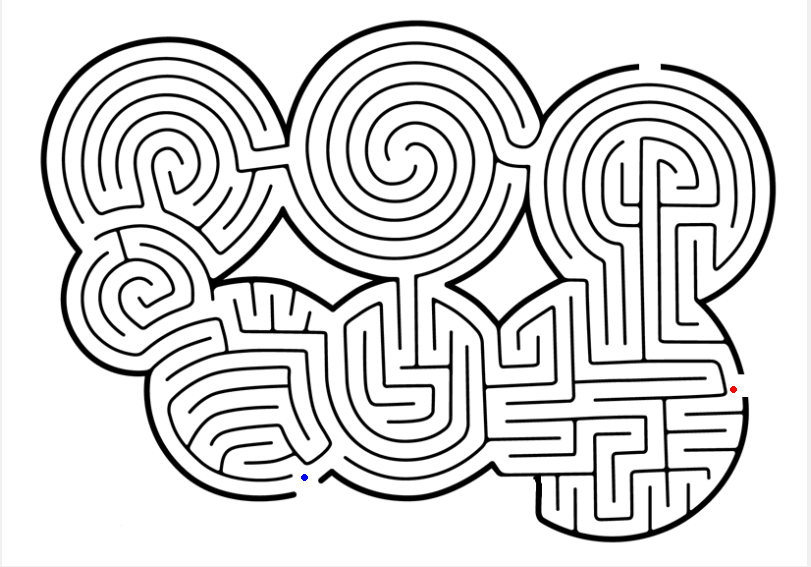
1. A single pixel distance from a wall did not suffice, as in some cases the edges had a “hole” of a single pixel, which is not at all visible in the image itself. Therefore , the solution would appear to “cut” through a wall.
2. A larger d value provided a nicer looking path, as the path is always at least d pixels away from any wall, and therefore appears more visible when displayed to the user.

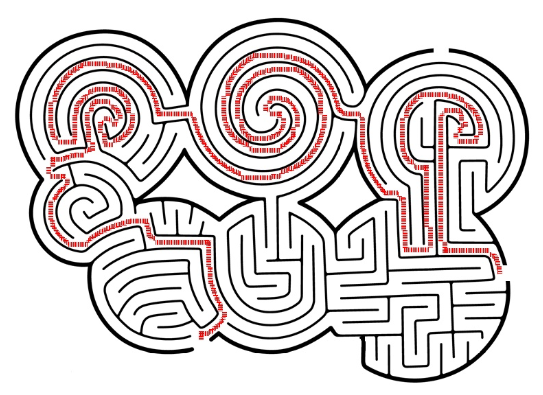
During the path search itself, we used a traditional BFS algorithm, searching for a path through the maze from start to end points. We used BFS in order to find the shortest path possible. the representation of the maze we performed BFS on was as such:

Also, for each Pixel , we keep it’s previous Pixel (the neighbor we arrived to from)

6. drawing the path: in case of a successful path discovered, we backtrack from the end pixel (using the previous pixel field) to find the path back to the start. During this process, we draw the path on the **original image** (original size and before cropping) by calculating the original coordinates of the pixels appearing in the path solution (using the reverse of the formulas previously mentioned).

In order to improve the path’s appearance, we thicken it in the direction that is farther from the wall (by d pixels at most to avoid drawing on the walls).







**Implementation:**

User Interface:implemented in c# using windows forms.

Maze solving script: implemented in python based on several methods and packages such as OpenCV, and NumPy.

**Results**

In the beginning of working on the Maze Solver project, our approach for a solution was mostly clear, using edge detection in order to discover the walls, use the edges map to create a graph like representation of the maze, and solve it with BFS. Our final solution indeed uses this basic approach, but we have discovered many challenges and issues we did not anticipate when thinking about the problem firstly. Some of the larger ones were:

Firstly, in our first version of solution, each pixel represented a vertex in the maze “graph” , but this caused issues as a single pixel is not so much visible in the original image, thus we encountered cases were path would go “through” walls. To solve this, we changed our definition of a wall vertex to a pixel that has a wall pixel in it’s d-surrounding. This change can be though of as changing the representation of the maze, to a new one where each vertex represent a square of d\*d pixels, instead of a single pixel.

Secondly, we encountered very slow runtimes which made us use the resizing and crop techniques to greatly reduce runtime , in the case where we indeed find a solution (in case of failure runtime can be slow because the algorithm would attempt to solve for each resizing and d values, but this was preferable in our opinion).

Thirdly, we encountered solution paths that went outside the maze, which we handled mostly by adding the image cropping (which required research on various techniques such as Morphological transformations ) and the cross edges.

With our final solution, we find that the majority of mazes and points we give the solver succeed in finding an optimal solution (excluding illegal points and very low quality maze images).

**Conclusions**

Our conclusions from working on this program, is that implementing a practical, efficient and stable application of image processing that works on all (or most) general examples is a very big challenge , as even relatively simple ideas get complex when needing to handle many different edge cases and complications.

We feel that our final result is pretty stable, and handles most mazes In a good way (big improvement on runtimes and accuracy)

We Do think that the application can be further improved in various fields:

1. Optimizations and runtime: runtime could be improved by introducing parallel programming when trying different parameters for d and resizing size.
2. Contour find- could perhaps be improved to a point where we can “crop” a more exact image of the maze, which will provide better runtime and avoid wrong solutions more efficiently.

**More Information**

**References**

Contour finding :

<https://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html>

Morphological transformations: <https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html>

On Mazes:

<https://en.wikipedia.org/wiki/Maze>

**References:**