Solving Mazes using Image Processing techniques and Basic Graph Theory

Neeraj Jadhav

September 2021

Abstract: *Mazes are quite common in day-to-day lives. Commuting between two points on a map is a basic example path finding techniques and application of Graph Theory. The difference in the two situations is availability of complete information of the scenario prior to travelling in case we have a map. This can be solved by the popular Watershed algorithm. We limit our discussion to the other situation.*
This paper aims to solve mazes using a randomised graph traversal algorithm keeping track of important checkpoints. It may not be the best possible solution but gives the correct result with any kind of maze.

Keywords: DFS, Thresholding, Dead-ends.

1 Introduction

First of all, the problem faced was to efficient, non-lossy conversion of the image to an array of grid-boundaries. Most of the algorithms need the maze to be in a format where the starting and ending pixels are marked with distinctive colors and path be of width 1 pixel. The images which are generally available may not be in this format. We approximate the image to a near-ideal requirement of the algorithm. Then, although it takes around 2× more time but the time complexity remains unchanged for most of the situations.

2 Approach

2.1 Pre-processing the image

We start by taking Adaptive Threshold on the initial image followed by a series of erosions and dilations for grouping the unevenly thresholded boundaries. This is followed by a set of iterations of Blurring (Gaussian blurring with a kernel of size 3). This smoothens the unevenness to a great extent.
Now, there are two approaches to reduce the white space as much as possible to keep the neighborhood as less as possible for higher efficiency. One is to resize the image by a factor obtained by taking the average path-width. Path-width can be calculated by drawing a series of Hough-Lines then taking the distance between two consecutive of these. But, this makes task tedious. For sake of simplicity we take use the other way. That is, we process the image again in a cycle of thresholding, erosion and dilation (the number of erosions are kept higher than dilations for reducing the path width). This is followed by placing a blue border frame in all four directions (ending positions) and a red colored pixel at the center (starting position).

2.2 Traversing

This part is relatively simpler. Whenever, we make a move we mark the previous pixel as green. Also, we reject green pixels from being counted as valid neighbors to prevent infinite loops. The only pixels which will be valid neighbors would be white or blue. And since the algorithm is just a depth first traversal, the only things we needed to take care of were dead ends and checkpoints.

- **Dead End:** If a pixel has no white or blue neighbor, it is a dead end

- **Checkpoints:** If we were able to make more than 1 valid move from a point, it is a checkpoint.

Whenever we reach a dead end, our strategy was to backtrack the checkpoints and start traversal in a new direction henceforth. Since we keep marking the traversed pixels with green, it automatically prevents retracing same path again. Finally, we store the valid path as a list and mark it at the end.

3 Case Study

3.1 Formal Description of the Approach : Wikipedia

Trémaux's algorithm, invented by Charles Pierre Trémaux, is an efficient method to find the way out of a maze that requires drawing lines on the floor to mark a path, and is guaranteed to work for all mazes that have well-defined passages, but it is not guaranteed to find the shortest route.
A path from a junction is either unvisited, marked once or marked twice. The algorithm works according to the following rules:

- Mark each path once, when you follow it. The marks need to be visible at both ends of the path. Therefore, if they are being made as physical marks, rather than stored as part of a computer algorithm, the same mark should be made at both ends of the path.
- Never enter a path which has two marks on it.
- If you arrive at a junction that has no marks (except possibly for the one on the path by which you entered), choose an arbitrary unmarked path, follow it, and mark it.
- Otherwise:
  - If the path you came in on has only one mark, turn around and return along that path, marking it again. In particular this case should occur whenever you reach a dead end.
  - If not, choose arbitrarily one of the remaining paths with the fewest marks (zero if possible, else one), follow that path, and mark it.

The "turn around and return" rule effectively transforms any maze with loops into a simply connected one; whenever we find a path that would close a loop, we treat it as a dead end and return. Without this rule, it is possible to cut off one's access to still-unexplored parts of a maze if, instead of turning back, we arbitrarily follow another path.

When you finally reach the solution, paths marked exactly once will indicate a way back to the start. If there is no exit, this method will take you back to the start where all paths are marked twice. In this case each path is walked down exactly twice, once in each direction. The resulting walk is called a bidirectional double-tracing.

Essentially, this algorithm, which was discovered in the **19th century**, has been used about a hundred years later as **Depth-First search**.

Note: The algorithm that we used is a variant of the Trémaux's algorithm. The difference being higher number of checkpoints. This results in more time being taken to solve the problem but eventually leads to correct solution keeping the logic simple.

3.2 Creation of Mazes

Below is a list of popular **Graph Theory** based algorithms to create mazes

- Randomized depth-first search
  - Recursive implementation
  - Iterative implementation
- Randomized Kruskal's algorithm
- Randomized Prim's algorithm
  - Modified version
  - Simplified version
- Wilson's algorithm
- Aldous-Broder algorithm

A maze can be generated by starting with a predetermined arrangement of cells (most commonly a rectangular grid but other arrangements are possible) with wall sites between them. This predetermined arrangement can be considered as a connected graph with the edges representing possible wall sites and the nodes representing cells. The purpose of the maze generation algorithm can then be considered to be making a subgraph in which it is challenging to find a route between two particular nodes.

If the subgraph is not connected, then there are regions of the graph that are wasted because they do not contribute to the search space. If the graph contains loops, then there may be multiple paths between the chosen nodes. Because of this, maze generation is often approached as generating a random spanning tree. Loops, which can confound naive maze solvers, may be introduced by adding random edges to the result during the course of the algorithm.

We will limit the discussion to the Depth-First Search algorithm here, others can be found out here.

Randomized DFS : We start with just a grid of squares and choose one of them as the starting point, then follow these steps eventually till we step out of the rectangular outer boundary

- Given the current cell as a parameter
- Mark the current cell as visited
- While the current cell has any unvisited neighbour cells
  1. Choose one of the unvisited neighbours
  2. Remove the wall between the current cell and the chosen cell
  3. Invoke the routine recursively for a chosen cell (or let it be iterative removal for simplicity) which is invoked once for any initial cell in the area