A Maze Solver for Android

Rohan Paranjpe
Department of Electrical Engineering
Stanford University
Stanford, CA 94305
Email: rap2363@stanford.edu

Armon Saied
Department of Electrical Engineering
Stanford University
Stanford, CA 94305
Email: armons@stanford.edu

***Abstract*—Solving mazes and extracting shortest path solutions have a number of applications within image processing. In this paper, we provide an algorithm and develop an Android application used to detect and solve rectangular mazes using region labeling, morphological thinning, and template matching. We provide a detailed analysis for our app's performance in four categories of simply connected mazes and achieve a high success rate with *canonical* mazes, those whose walls are approximately as thick as their path space. We provide a discussion on some interesting applications of our algorithm and shortest path problems within image processing, and improvements upon the algorithm to increase its robustness and success rate.**

I. Introduction and Motivation

We encounter many different kinds of puzzles everyday and often need the shortest path between two points or a solution. Our motivation behind this project was to create an application which can automatically solve a maze for the user. Maze solving and shortest path algorithms within image processing are very important in a number of different applications ranging from route mapping to feature extraction and seam carving within images [1]. In this paper, we present an image processing application which obtains an image of a maze, extracts the solution space, solves the maze, and then overlays the solution over the original image.

Mazes in general are extremely varied in terms of complexity, topology, and shape, so we first need to restrict our space of mazes to those which are simply-connected, meaning there are only two disjoint walls in our maze, and those which have their starts and ends on their edge. Note that within this definition we do not require a strictly rectangular maze, but that for the most part, rectangularly shaped mazes will be our primary type of maze by default. We use these restrictions to limit our mazes to those with no loops, and those whose starts and ends are detectable via image processing.

Our goal was to create a mobile application which does all processing on the phone (no server-side communication), and that can detect unusual-looking mazes, but only those that follow our criteria above (e.g. hand-drawn mazes, or aerial views of hedge mazes). We used the Android mobile platform and OpenCV toolbox for common image processing functions ranging from adaptive thresholding and erosion to more complicated morphological operations such as region labeling and convex hull calculations.

The paper is organized as follows. Part II will cover Algorithm Development, including justification of our steps and processes. Part III will cover our results and observations with different test cases. Part IV will include our conclusions, further improvements, and relevance within image processing.

II. Algorithm Development

Our algorithm is comprised of a series of steps to perform image capture, binarization, filtering, maze detection, solving, and overlaying the solution. The steps are summarized in the block diagram in figure 1. The following sections outline in detail the process behind each step.

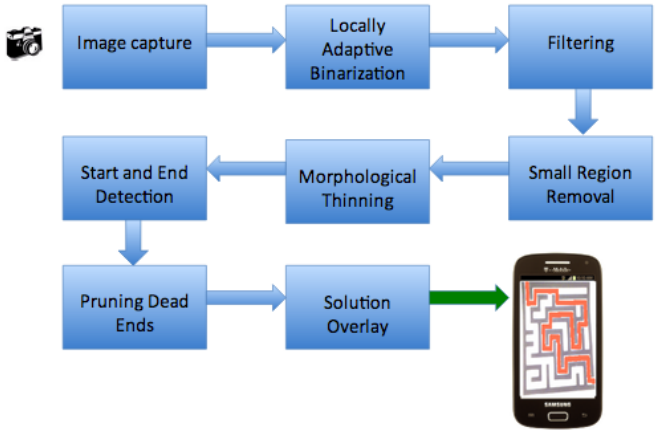


Fig. 1: Flowchart of the algorithms used for processing the maze

A. Image Capture and Filtering

Although we initially ventured to create a real-time solver, due to computational constraints and the difficulty for the user keeping their hand still long enough for the processing, we decided to use a still image capturing method and proceeded to process the captured image rather than a video.

Our first step was to use locally adaptive thresholding to binarize the greyscale image using an OpenCV implementation with a 3-by-3 window. This method slides the window over the image and calculates local thresholds using Otsu's method [2]. We used this method rather than a global threshold in order to reduce the effects from uneven lighting, shadows, and other noise. At this stage we have a binary image of the maze with numerous contours representing the maze, the edges of objects outside of the mazes bounds, shadows, and other noise.

For small noise reduction at this point, we applied a 3-by-3 median filter. This worked for small effects such as salt-and-pepper noise or random fluctuations in the image. Our next step was to detect the maze itself by region labeling within the image. We used a region labeling function in OpenCV to label the various connected regions within our black and white image. Given our restriction of the maze space, we knew that our maze would consist of two walls. Our next assumption lied within the idea that the maze walls, while perhaps thin and small in area compared to other objects in the image, would certainly have large perimeters. And so we identified the two regions with the largest perimeters as the walls of the maze, and set the remaining regions to the background.

At this point in the process our image was nearly noise-free, and comprised of the maze walls in black, with the rest of the image in white. Our next step was to filter out everything but the maze, and only operate on the solution path. Initially, we tried using a binary mask built from the convex hull of the maze walls, but this was not robust to slight rotations or geometric shrinking/dilation of the maze image. As a result, the convex hull of the regions would leave strips of white along the edge of the maze walls which could later contribute to a bad start/end detection scheme.

To rectify this, we used a slightly different method to extract a better binary mask, which would include only the maze walls and the solution space. In order to correctly extract this mask, we needed to mathematically define points which fit these criteria.

With two maze walls, we are essentially trying to find the set of all convex combinations between these two walls. Mathematically, let W_1 be the set of all (x, y) points within the first wall, and let W_2 be the set of all points within the second. Define the set A as:

$$A = \{(x,y) | (x,y) = \lambda(x_1, y_1) + (1-\lambda)(x_2, y_2), \\ (x_1, y_1) \in W_1, (x_2, y_2) \in W_2, 0 \leq \lambda \leq 1\}$$

Calculation of A required a bit more computation, but ensured a binary mask free from sliver artifacts. After applying the mask, as a final filter, we used region labeling to extract small regions, so that by the end of our preprocessing, we were left only with a binary image, with the solution path in white, and the background in black.

B. Morphological Thinning of the Solution Space

Our next step was to use morphological thinning to obtain a full skeletal path of the solution space. This was important for a number of reasons: to simplify our solution search, to prune the non-solution path, and to simplify obtaining the final solution from the path space. But in order to obtain a usable skeleton, we required a thinning process that would leave us with the following features:

- one-pixel wide
- completely connected
- no loops

The one-pixel width requirement is for facilitating our pruning process (pixelating the wrong paths). There are lots of quick thinning algorithms that do not guarantee that a region gives a connected skeleton. In the scope of our project, if a particular path was to have a gap within the skeleton, this would mean we could never extract a solution that would run through that path. After trying out various algorithms, we finally used the Zhang-Suen thinning algorithm, which guarantees connectivity and a one-pixel wide solution [3]. Unfortunately, even this algorithm does not completely satisfy all of our requirements for a proper pruning algorithm, and our further developments will be elaborated on in the "Pruning and Solution Overlay" section.

Once we obtained a workable skeleton, we had to classify particular pixels within the maze as members of the following sets: endpoints, junctions, and path pixels. Endpoints are pixels with only one neighboring white pixel, junctions are pixels with strictly more than two white neighbors, and path pixels are pixels with only two neighboring white pixels. We used 8-connectivity to measure the number of neighbors for a particular pixel (this includes diagonal as well as horizontal and vertical neighbors). The starts and ends of the maze are also deadends, and so start/end detection can be classified as a problem where we need to find the most likely two points within our set of endpoints that are the start and end of the maze.

C. Start and End Detection

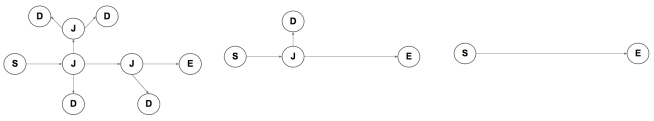
Detecting the start and end of the maze was a somewhat difficult task in practice. We developed several algorithms to detect the start and end points of the maze, all of varying effectiveness. Our final implementation comprised of the following steps:

1) Using a binary mask to acquire the outermost regions of the maze
2) Using a ranking function on each endpoint to determine its likelihood of being the start or end.
3) Extracting the two highest value endpoints according to the ranking

To determine the regions of the start and end points we used the fact that the entrance and exit to the maze lie on the maze's edge. By creating a bounding box of the maze walls we were able to acquire a minimum-bounding rectangle with a border of some variable width. We determined this width empirically; a width that's too small for the mask would not pick up many pixels, but a width too large might pick up pixels that are not in the start and end regions, but within the maze itself. After determining an appropriately sized mask, we obtained a binary image with a large number of pixels in the start and end regions, and a smaller number of pixels in other regions. We define a set W which contains all white pixels after this operation.

Obviously, the start and end of the maze will lie close to or within the regions with a large number of pixels. In order to determine an endpoint's likelihood of being a start or end, we used the following ranking function:

$$f(i) = \sum_{j \in \mathrm{W}} e^{-r_{ij}/r_0}$$



(a) I=0 (b) I=1 (c) I=2

Fig. 2: Shown above is the maze in graph form with S,D,E,and J representing the start, dead ends, end, and junctions of the maze. Each graph shows a successive iteration with the algorithm until we are left only with a solution path from the start to the end. I represents the number of iterations that have taken place.

where r_{ij} is the Euclidean distance between an endpoint i and one of the white pixels j, and r_0 is a scaling factor for the exponential. We can see that an endpoint located very close to a cluster of white pixels will obtain a high value, and one farther away obtains much less weight. This works very well for separated start and end points, and helps eliminate incorrect endpoints. There are however, some faults to this method. One is that if the skeleton produces two endpoints that are in the start or end region (which can happen as a result of the thinning algorithm), we may classify both of these as the start and end of the maze (which would result in a very silly looking solution). There are ways to fix this: one could invoke clustering to try and select one representative endpoint from a region, and one from the other, but for most of our solutions, this sum of exponentials ranking worked fine (even in cases where the start and end were located next to each other in the maze).

The final step simply took the highest valued endpoints and extracted them from the list of endpoints.

D. Pruning and Solution Overlay

Our final post-processing step was to iteratively prune endpoints from the skeleton until we were left with an unchanging skeletal path. Mathematically, this lies on a graphical assumption about the maze, namely that endpoints will never be connected to other endpoints, and will only be connected to junctions. Figure 2 shows graphs of junctions, endpoints, start and ends and the iterative process taking place. If we start pixelating by starting from the endpoints, we can iteratively remove parts that are not part of the solution space (reclassifying junctions that lose their neighbors as endpoints during the process), until no more endpoints exist. This non-recursive implementation was important for producing the desired results quickly and without requiring too much memory.

Unfortunately, there are particular skeletons that are unsolvable for our pruning algorithm. Specifically, if the skeleton produced contains two or three connected junctions, it cannot be pruned correctly. As an example, L-junctions or T-junctions can produce unpruneable structures. This is why, in addition to Zhang-Suen thinning, we had to employ a template matching filter to extract L and T junctions from the maze as well. Our final post-processing steps were to take the pruned skeletal image, use dilation in order to thicken the solution, and overlay it in red over the original color image.

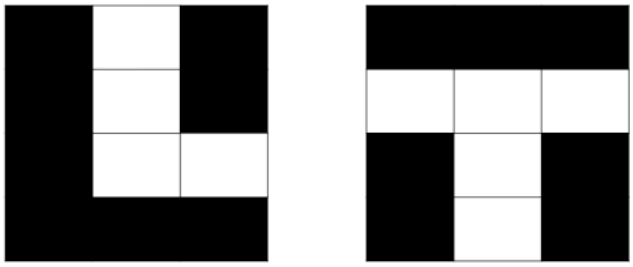


Fig. 3: Two examples of stable junctions under our pruning algorithm. To correct for this, we can pixelate the corner pixel in the L-junction, and pixelate the top-middle pixel in the T-junction.

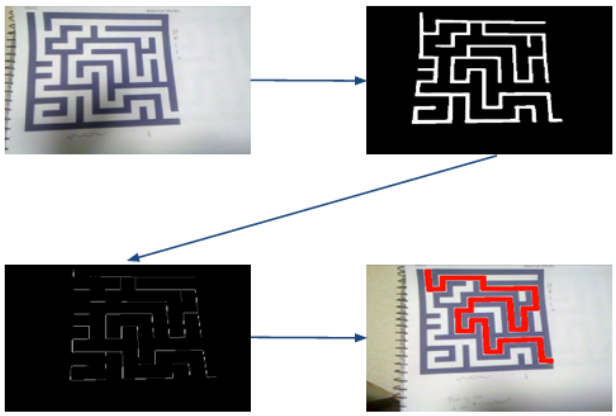


Fig. 4: The transformation of the image during each step. Note that due to rendering errors on the Droid phone, the third image (morphological thinning) shows gaps.

III. Results and Observations

Our algorithm and app worked well for a variety of mazes, and not so well on others. For analysis, we split our group of mazes into four different categories, canonical, thin-walled, complex, and miscellaneous mazes with irregular shapes, objects, etc. We define each kind of maze in the sections below and describe our solver's performance for each category. Figure 5 shows mazes solved from each category.

A. Canonical Mazes

For our purposes, canonical mazes are defined as those with walls whose thickness is comparable to that of the solution space thickness, and that lack noise in terms of objects that are within the maze (drawings, lettering, etc.). Furthermore, we restrict the complexity of the canonical maze as those whose maximum depth is 2. This means that the canonical maze can be solved entirely within 2 iterations of our pruning algorithm.

For this set of mazes, our app worked very well, achieving a relatively high success rate. Of the 10 mazes we tried, it was able to perfectly solve 9 of them, with one maze that showed the full solution, but also included a path that wasn't part of the main solution. There were a number of different factors that contributed to the performance, namely binarizing

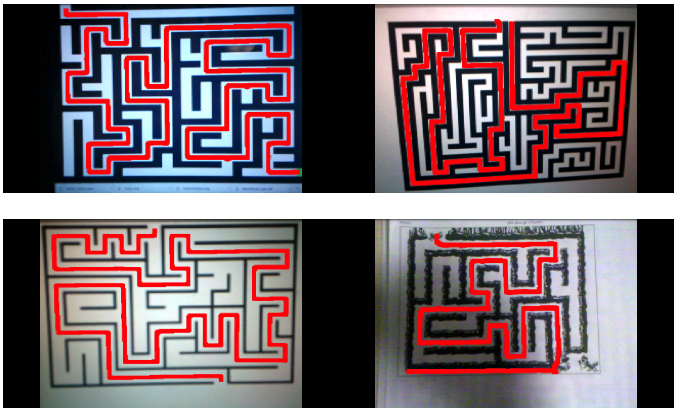


Fig. 5: Shown above are four mazes from each of our defined categories. The top-left, top-right, bottom-left, and bottom-right show our application's solution for a canonical maze, a complex maze, a thin maze solved, and a maze from the miscellaneous category respectively.

the maze and detecting the start and end correctly. With the algorithms we implemented, canonical mazes were generally solved pretty easily and errors were usually within the actual solving algorithm or were correlated with a grouping of two or more connected junctions.

B. Complex Mazes

Complex mazes are similar to the canonical maze, except that their maximum depth is strictly greater than 2. From an image processing standpoint these mazes are of the same difficulty as the canonical maze. However, because of their depth, a lot more pruning needs to take place before the solution can be shown. Because of this, there is a higher probability of error and looped junctions, and so the app does not always solve complex mazes. Often, we found that changes like the orientation of the maze (by 90 degree rotations) could alter the final solution. We attribute this error to the occasional introduction of loops due to the Zhang-Suen algorithm not preserving homotopy [4].

There are methods to fix this as well, and we will expand upon some of these ideas in the conclusion. For the most part, we were able to obtain full solutions for most complex mazes (although not on the first try at all times). Usually we obtained either partial or full solutions with extended non-solution paths for this category of mazes.

C. Thin-walled Mazes

We define thin-walled mazes as those that have walls of thickness significantly smaller than the width of the path space. We chose this category to demonstrate the limitations of our algorithm. Thin-walled mazes can present problems for primarily two aspects of our implementation.

First, recall that our method of start and end point detection defines a bounding box of the maze space and uses a heuristic of shrinking the box dimensions by a predefined number of pixels. This process pinpoints the entrance and exit of the maze

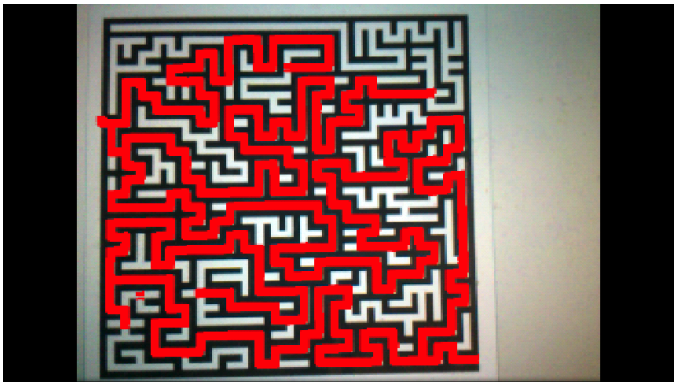


Fig. 6: This is a maze that was solved, but still retains non-solution paths in the structure. We classify this as a "partially solved" maze.

since the edges of the masked image will be black uniformly except at the desired locations. Given the constraints of this method, our application struggles to define the start and end of mazes with sufficiently thin walls. However, if the maze walls are just slightly thicker our algorithm succeeds in solving the maze as in figure 5.

Secondly, certain thin-walled mazes can present a problem for our implementation of morphological thinning. As mentioned earlier, the Zhang-Suen thinning algorithm we used can potentially produce loops in the skeletal structure when there are large-width paths present near multiple junctions in the solution space. This can lead to a malformed skeletal path with stable junctions that cannot be pruned.

D. Miscellaneous Mazes

This group of mazes is essentially a handful of "non-canonical" mazes on which we wished to test the extent of our application's capabilities. To accomplish this, we created hand-drawn mazes and found several children's mazes with extraneous objects in the margin and occasionally in the solution path. These mazes were characterized by some of the following qualities: having non-solid walls, non-rectangular shape, small objects near and/or in the maze itself. Our application solved nearly all of the children's mazes we tried. The small component filtering algorithm succeeded in removing the extra non-maze features and our region labeling was able to identify the solution path.

However, hand-drawn mazes presented a problem when they were created with walls of varying thickness, thin walls (as described previously), and extremely non-perpendicular shapes. "Rounded rectangular" shaped mazes, for example, were not difficult for our algorithm to solve given that the walls were sufficiently thick. The limitations of Zhang-Suen popped up here again when large solution spaces thinned to regions with loops and became unsolvable.

Overall our algorithm performed well within the confines of our defined input domain, and not as well with those of varying thicknesses and sizes for the walls and solution spaces.

IV. Conclusions and Further Work

Overall our app and algorithm worked well on a variety of different kinds of mazes. One recurring problem we had was the Zhang-Suen thinning algorithm's inability to preserve homotopy. This meant that skeletal paths would often introduce loops at some junctions, which is stable under the endpoint pruning algorithm. There are other ways to deal with this, namely a search of the solution space and loop removal, or a modified algorithm which preserves homotopy. However for the most part, the thinning performed well, and in conjunction with L and T junction template matching, was able to achieve a relatively high success rate especially within our group of canonical mazes.

For robustness, we employed a series of measures to avoid redundancy in the algorithm and code. For example, computations over the image were often restricted to the white pixels (usually the solution space), which makes up around 2% of the total image at times. Other computations, such as the sum of exponentials in the start/end detection were surprisingly quick because of this method.

In order to improve our application's robustness in detecting input images we could add functionality to detect the skew angle of the maze. Because our algorithm for start and end detection relies on receiving an approximately perpendicular input maze, the application is somewhat sensitive to rotation. Although the convex combination of the walls provides a good noise reduction to prevent this, it is not completely foolproof, and an angle detection coupled with subsequent rotation of the input image would have been a useful modification. Unfortunately we were unable to fully implement this scheme due to time constraints.

One very big computation lied in obtaining the convex combinations of the wall as explained above and in the "Algorithms" section. This computation was done for each pixel in the image to create the mask, and there are certainly better ways to find this set of pixels. One involves simply taking the convex hull of the walls (much quicker computationally), but this leaves white sliver artifacts which are hard to detect and extract, especially because they are at times connected to the solution space itself.

Although our maze-solving application for Android was largely successful for the scope of mazes we defined, it is worth questioning how this project can be extended to other more practical applications. One interesting realm of problems is in path finding, specifically with regard to topographic maps. Topographic maps represent elevation with contour lines, which can be processed using similar algorithms presented in this paper. It is possible to imagine a scenario where someone requires the shortest path or minimum energy expended through some region with varying elevations.

In this case, our maze-solving algorithms can be extended to process the image of such a map, apply region labeling to different contours as level sets, and then produce a maze-like representation of the map [5], [6]. Mazes actually fall into a special class of topographical maps with only two level sets; the walls represent a set with infinite cost for traversal and the solution space represents a level set with zero cost for traversal. This representation would define costs for regions of elevations exceeding a particular threshold desired by the user. This kind of work is already being conducted and projects such as ours provide a springboard for further research in this exciting field.

Acknowledgments

The authors would like to thank Peter Vajda and David Chen for their help and guidance throughout the project. They would also like to thank Bernd Girod and the TA's for EE 368 and a great quarter.

Appendix

Both authors contributed to algorithm development. Rohan ran implementations of the solution search within MATLAB and Armon helped port over implementations into the Android software and created the code structure used for the app. Both authors contributed equally to the poster and paper.

References

[1] S. Avidan and A. Shamir, "Seam Carving for Content-Aware Image Resizing", *ACM Transactions on Graphics*, Vol. 26, No. 3, Article 10, 2007

[2] N. Otsu, "A Threshold Selection Method from Gray-Level Histogram", *Systems, Man and Cybernetics, IEEE Transactions*, Vol. 9 , Issue 1, 1979

[3] T. Y. Zhang and C. Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns", *Communications of the ACM*, Vol. 27, No. 3, 1984

[4] J. N. Wilson, G. X. Ritter, "Handbook of Computer Vision Algorithms in Image Algebra", pg. 166, 2010

[5] S. Jung and S. Pramanik, "An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps", *IEEE Transcations on Knowledge and Data Engineering*, Vol. 14, No. 5, 2002

[6] A. Khotanzad and E. Zink, "Contour Line and Geographic Feature Extraction from USGS Color Topographical Paper Maps", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 25, No. 1, 2003