Solving Irregularly-Shaped Mazes Using Cellular Automata With State Machine Logic

Justin A. Parr
q3ej7ejsjb@snkmail.com
http://justinparrtech.com
Version 1.1, 3/2/2016

Executive Summary

Existing maze-solving algorithms require ideal conditions or environments, such as crisp angles, ideal contrast, and uniform proportions.

This paper specifies the details of a flexible and efficient algorithm that is capable of analyzing and solving any hand-drawn maze that follows certain basic rules.

Table of Contents

Version History

Version	Date	Author	Revision Notes
1.0	2/2016	Justin Parr	Initial Release
1.1	3/2/2016	Justin Parr	Fix Typos

Overview of Existing Solutions

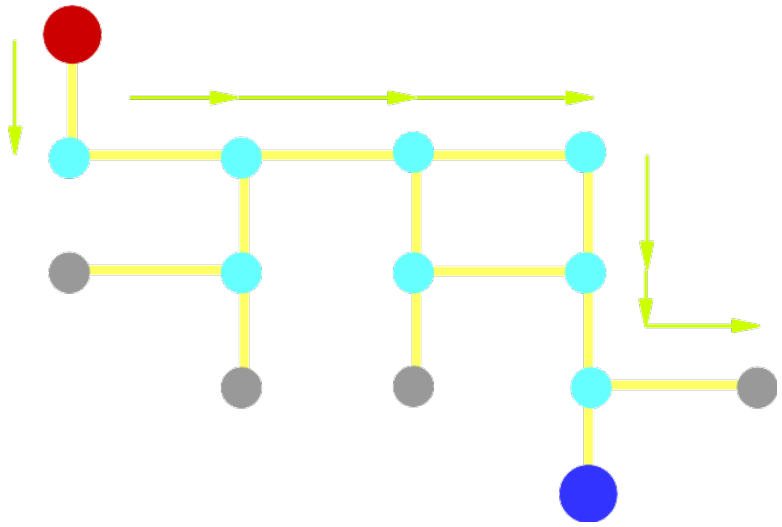
Node / Graph Analysis

A node map, or graph, shows the relationship between all relevant points in a maze, with redundant or irrelevant detail eliminated.

Relevant points are start, end, junctions, and dead-ends, and the pathways of the graph show how they interconnect, along with a distance (or cost) metric.

The algorithm itself uses a polynomial or recursive approach to find the best path from start to finish, through the graph.

A graph-based solving algorithm must either have node and path information provided from another source, or it must be able to generate node and path information.



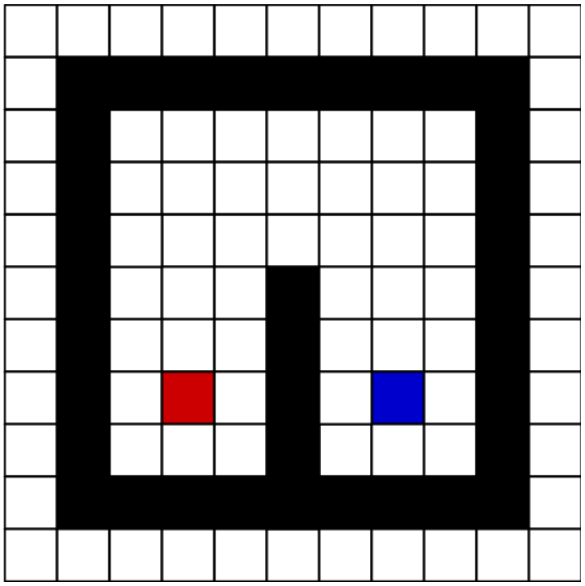
In the real world, GPS navigation systems are an example of a node-based solving algorithm – these systems use a GIS (Graphical Information System) database, to map node and path information, plus metrics such as distance, maximum speed, and typical traffic impact, to a bitmap or outline view of a road map. The solver algorithm crunches all of the node and path information in order to find the most efficient route between two points.

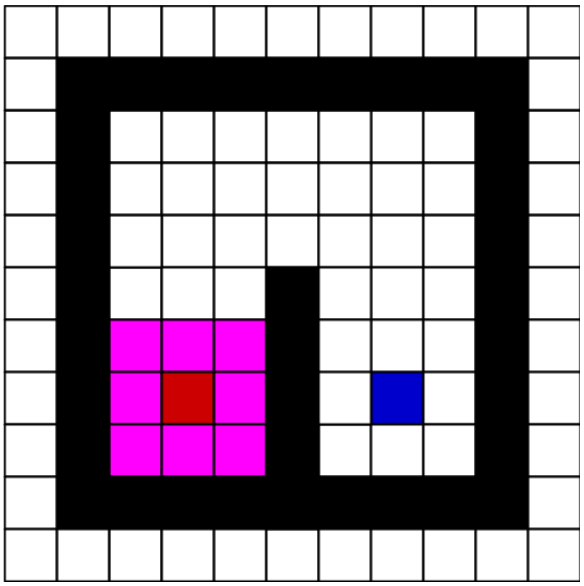
(Courtesy of Google)

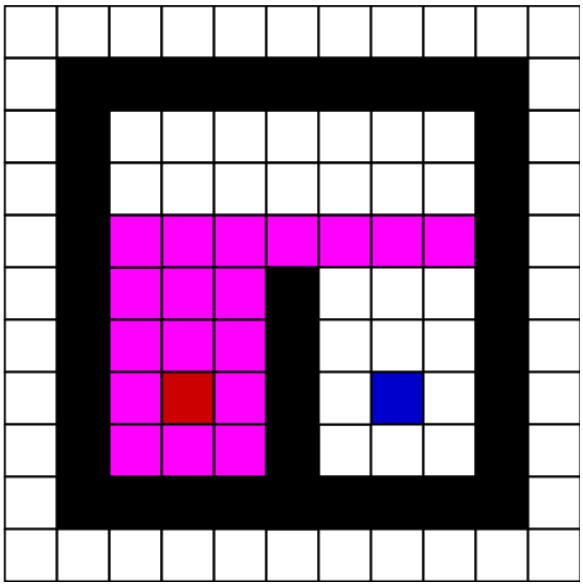
Flood Fill

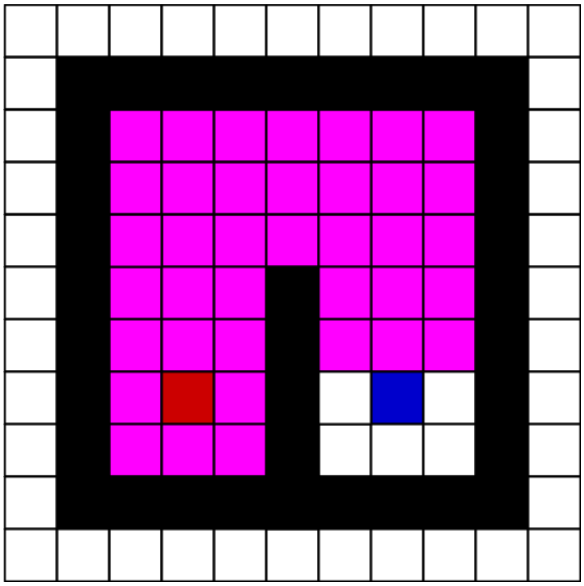
Flood Fill algorithms do exactly that – they flood a map, progressing from start to finish, until the finish point is reached.

Using a metric, some portion of the result set can be eliminated, leaving a vague path from start to finish.







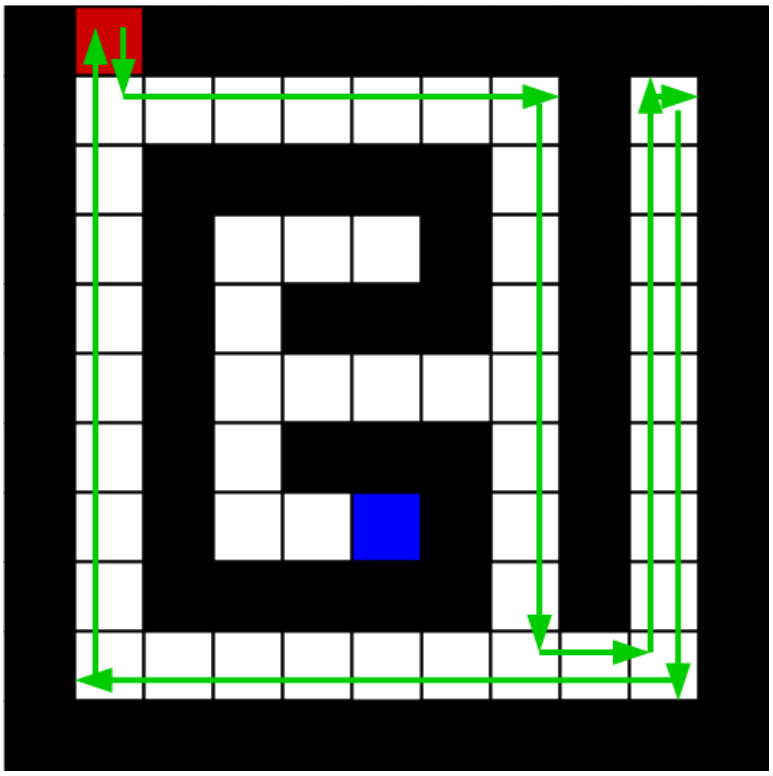


Flood fill algorithms are very flexible, and can be used in a variety of applications, but they don't produce a discreet result set – rather, the result set produced by a flood fill contains multiple possible solutions, including an ideal solution.

The figures above demonstrate an extremely simple maze, and flood fill.

Left-Hand Rule

A mathematical approach to maze-solving is the left-hand rule. This uses a mathematical concept called *topology* to guaranty that you will enter AND exit a maze, by simply placing your left hand on the left wall of the maze. As you turn corners, etc, you keep your left hand on the wall.

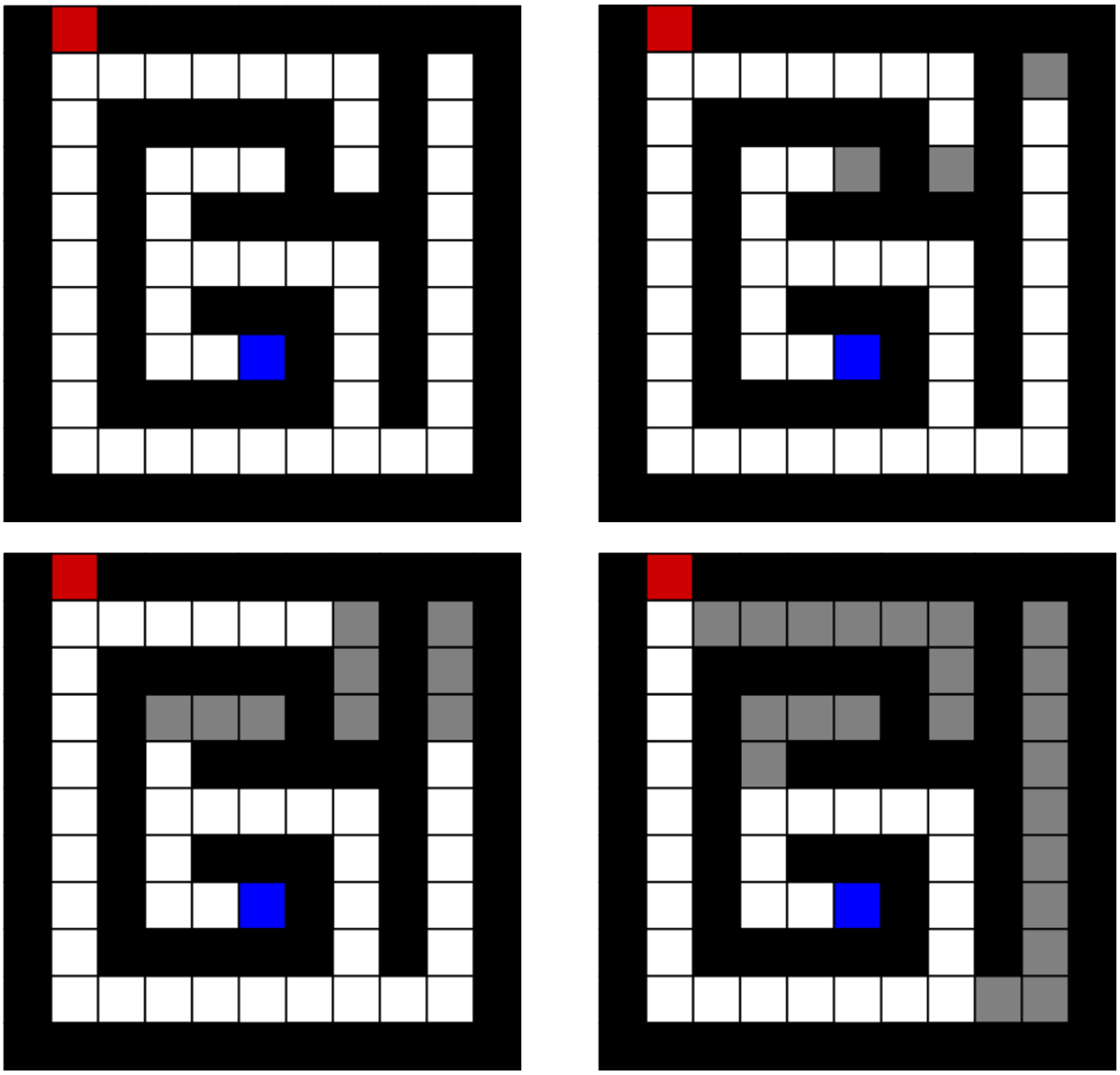


The problem with this approach is that it doesn't guaranty that you'll reach your *objective*. A certain class of mazes can be constructed, using concentric loops, that defy the left-hand rule. Because each layer is physically-isolated from the previous layer, you'll never reach the inner layer.

In the figure above, we can see that the “left hand rule” results in a complete circuit of the maze, returning to the starting position, without ever considering the “inner layer” of walls that happens to contain the objective.

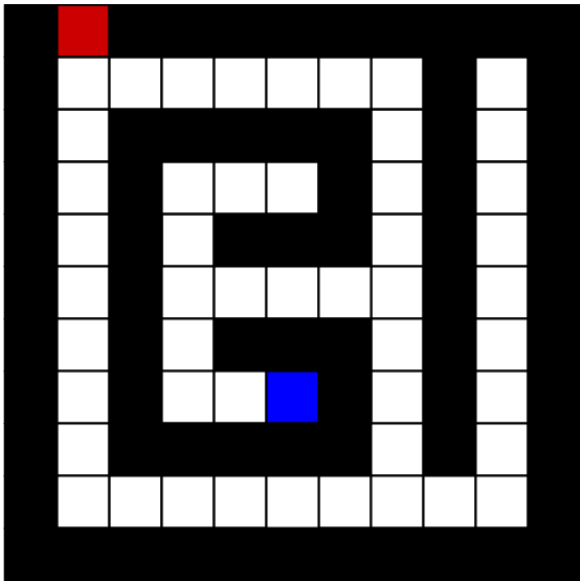
Existing Cellular Automata Solution

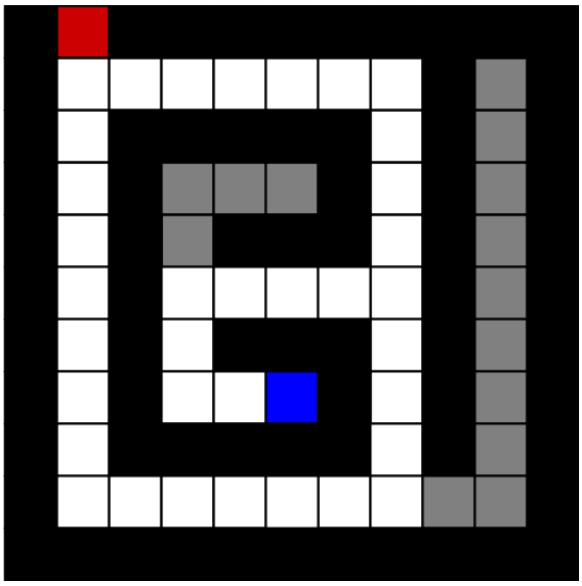
Maze-solving solutions based on Cellular Automata already exist – they use the concept of dead-end filling, to carve away all of the "invalid" paths, leaving a single, remaining path.



This works effectively, but requires ideal conditions. A pathway must be exactly the width of one cell for this approach to work. In the figures above, we see an example.

As with the left-hand rule, this approach can't easily solve mazes with multiple solutions, as we see in the figures below:





Background Information

This solution makes extensive use of State Machine Logic, and Cellular Automata, which itself is based on State Machine Logic.

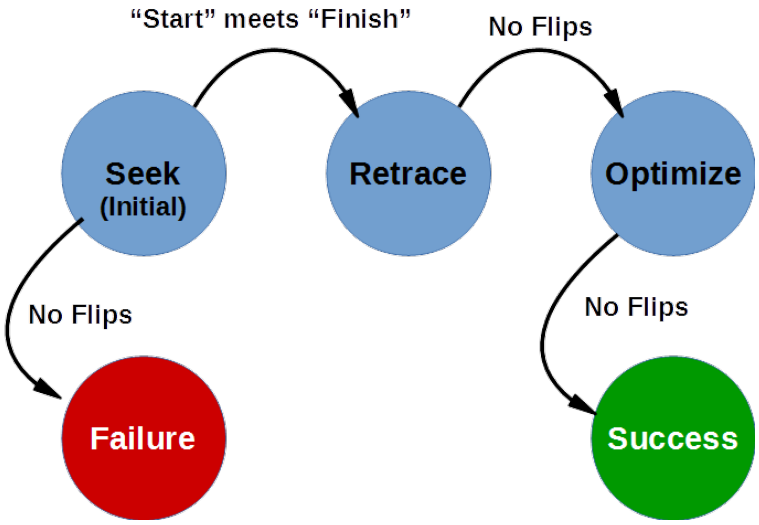
For more information about State Machine Logic, read this:
https://en.wikipedia.org/wiki/Finite-state_machine

For more information about Cellular Automata, read this:
https://en.wikipedia.org/wiki/Cellular_automaton

State Machine Logic

State machine logic leverages the concept that the entire system exists as a "machine" that is in a specific state.

The machine can switch states based its current state, specific input, and the specific set of rules for the current state.



A state machine reads its input, evaluates that input based on the rules associated with its current state, then finds a new state, and repeats the process until an "end state" is reached.

In addition to the cellular automation logic, in this solution, we use a larger system of states to represent machine "phases" or modes, that seek to accomplish a specific portion of the overall task.

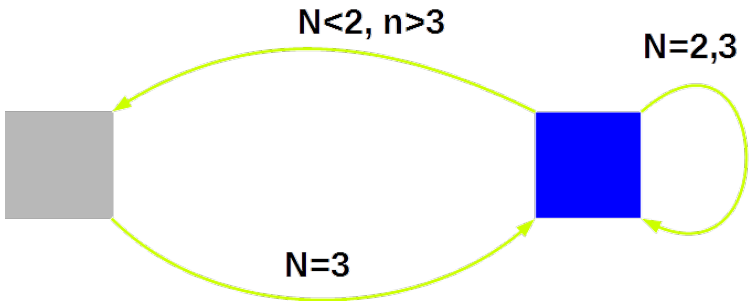
The diagram above is a state machine diagram for our application.

Cellular Automata

Cellular automata are a type of system that breaks an environment up in to cells that are arranged in to a grid.

Each cell exists in one of several possible states, and may store other specific information, such as a distance metric.

State machine logic governs how a specific cell reacts based on summary knowledge of its neighbors states.



In the most famous example, known as Conway's Game of Life, a simple set of rules allows each cell to react based on the count of its neighbor cells. The state machine for Conway's Life is depicted above.

- Each cell is either “alive” or “dead” (one of two states), depicted above by blue (alive) and gray (dead)
- In a dead state, if a cell has exactly 3 neighbors, it transitions to a “live” state
- In a live state:
  - If a cell has 2 or 3 neighbors, it persists in a live state
  - If a cell has less than 2 neighbors, it dies of loneliness
  - If a cell has greater than 3 neighbors, it dies of overcrowding

These simple rules, when seeded with either random values, or specific known arrangements, yields incredible complexity.

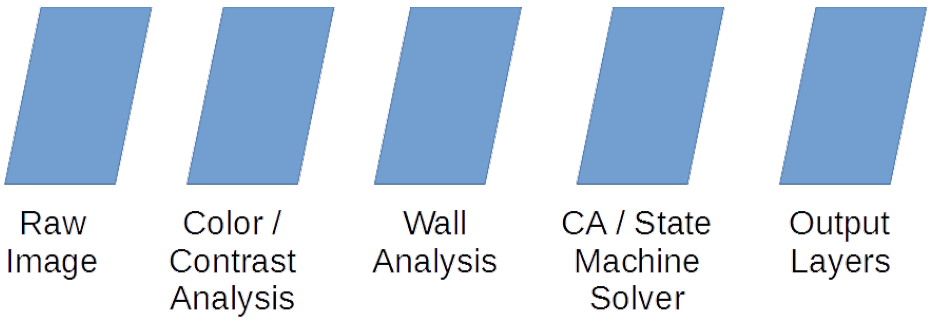
In our solution, we will use specific cell states to represent specific portions of the overall solution, and each cell will change states and respond to a set of rules within an overall state machine framework.

Project Goals

The following are goals of this project. **The solution must:**

- Interpret a bitmap of a hand-drawn maze as follows:
  - Drawn on a white sheet of paper – white spaces represent open areas that may or may not be part of the solution set, and are capable of changing states.
  - Maze walls are drawn with a black marker – black spaces represent immutable obstacles
  - The maze start is drawn as a red dot – the "most red" cell is the start of the maze
  - The maze finish is drawn as a blue dot – the "most blue" cell is the maze finish
- Leverage simple image analysis to convert a raw bitmap in to a weighted cell array that can be fed to the solver algorithm.
- Be computationally-efficient.
- Ignore shape, scale, and the requirement to define nodes.
- Produce a discreet set of coordinates that represent cells along the path of the solution. The path must be one cell wide, and the coordinates represent a list of left-right-zero turns that result in navigating from "start" to "finish".
  Stated another way, starting at the first cell in the set, each subsequent cell is connected at 0 degrees, +90 degrees, or -90 degrees, representing straight ahead (to the next cell), a right turn (and forward to the next cell), or a left turn (and forward to the next cell), respectively.

Solution Overview



The solution design uses layers to process information in different ways, and then pass the results to adjacent layers.

- Acquire Image – This is the raw bitmap as acquired by the camera
- Color / Contrast Analysis – Breaks the image in to cells, finds the start (red) location and the finish (blue) location, and assigns initial cell color values.
- Wall Analysis – Looks at local contrast to find walls, and assign “wall” or “empty” initial cell states.
- CA / State Machine Solver – Iteratively runs the state machine logic until the maze is found, or until it is determined that no solution is possible.
- Output Layer – In the prototype application, this presents a visual representation of the solution, by tracing “path” cells over the original bitmap image. The output layer could also be used to store the final set of “path” cells as a data set within a file.

From a user's perspective, they see the following:

1. Acquire an image using the camera
2. Output of the initial analytics (color / contrast and wall analysis steps). Although this is technically only a point of interest, and irrelevant to the final output, it's both visually-interesting, and allows the user to confirm that the program correctly interpreted the parameters of the source maze.
3. The solver runs in realtime, ending when the maze is solved or the machine determines that no solution is possible.
4. The program presents the final solution, a set of “path” cells superimposed on the original bitmap

Solution Details

Image Analytics

Image analysis consists of three layers:

1. Acquire the bitmap
2. Color / Contrast Analysis
3. Wall Analysis

We'll look at each of these in detail.

Acquisition – Layer 1

As this is an Android application, acquisition consists of presenting a user interface with a preview window, where the user can activate the flash if required, ensure that the image is centered and focused, and take the picture.

The picture itself is saved to SD card as a JPEG, and then read back in as a bitmap object.

The bitmap is scaled to the screen resolution using the following method (assumes landscape mode, where device width > device height):

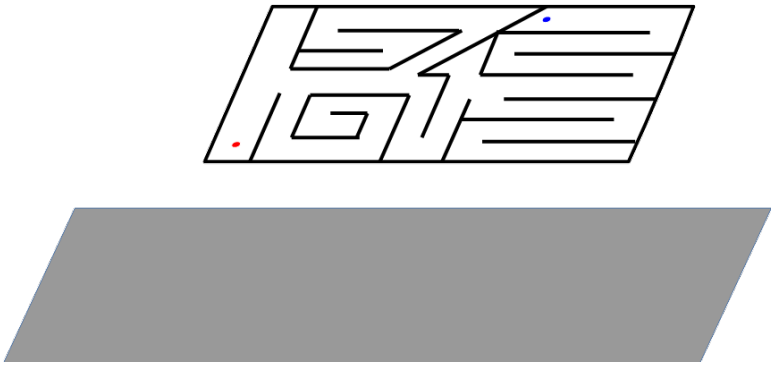
```
br = bitmap.width / bitmap.height

bh = device.height

bw = bh * br

if bw > device.width then
      bw = device.width
      bh = bw / br
```

- br – contains the bitmap's aspect ratio (width / height)
- bh – projected bitmap height. Initially, this matches the device height, assuming an aspect ratio the same or less than the device's aspect ratio.
- bw – projected bitmap width. Initially, we multiply the display height (bh) times the aspect ratio to obtain a proportional width.
- If the resulting width (bw) is greater than the device width, then we know that the picture is very wide, but short. We set the projected bitmap width (bw) to the device width, and then divide by the aspect ratio to obtain the proportional height.

This could be done with an if then / else, comparing the aspect ratio of the bitmap to the device, but in most cases, the camera's aspect ratio will be the same or less than the device aspect ratio, because the

camera takes images in standard formats, such as 16:9, but the device display is usually much shorter, such as 2:1. This means that the initial calculations for projected height and width will be correct ***most*** of the time, thus making it more efficient to assume that up front, and simply verify with an if then.

We start with a neutral gray field, and then overlay the scaled bitmap, centered, using the bitmap scaling function.



The result is a centered bitmap with gray borders.

Gray is used as the surrounding color, to provide neutral contrast for color and wall analysis.

Next, the cell size is calculated based on the canvas (scaled bitmap) height, bh:

```
cellsize = bh / 120
```

120 is an experimentally-derived constant, allowing sufficient resolution to distinguish maze features drawn in marker and then digitized in to a bitmap, while maintaining computational efficiency. As the number of cells increases, the number of pixels per cell decreases, and vice-versa. Larger cells are more computationally-efficent, but yield lower overall resolution, and can miss significant details. Smaller cells are more accurate, but less computationally-efficient.

The resulting cell size is used to aggregate individual pixels during the subsequent analysis layers.

Pixel Analysis by Cell – Layer 2

The bitmap is broken up in to groups of pixels, cellsize x cellsize.

For each pixel in the group, we perform three calculations:

1. Split the pixel in to red, green, and blue values

```
c     = pixelcolor
```

```
red   = (c and 0xff0000 ) / 0x10000
green = (c and 0xff00 ) / 0x100
blue  = (c and 0xff )
```

The raw pixel color consists of a 32-bit value, where the highest-order 8 bits are the alpha level, and can effectively be discarded.

The next 8 bits constitute the "red" value, then "green", and finally, "blue" is the 8 least significant bits.

"Why aren't you using YCrCb?"

Because, even though JPEG uses YCrCb color space, it gets converted to RGB when loaded in to a native bitmap object. We need a computationally-efficient way to take RGB color data, which is easy to obtain from a native pixel color value, and derive the various flavors needed for cell-based analysis.

2. Aggregate red, green, and blue values for the cell

```
totalRed   = totalRed   + red
totalGreen = totalGreen + green
totalBlue  = totalBlue  + blue
```

Aggregate values will be turned in to averages at the cellular level.

3. Find the darkest pixel value

```
contrast = red + green + blue
if contrast < darkest then
   darkest = contrast
```

Technically, this should be:

```
   contrast = sqrt( red * red + green * green + blue * blue)
```

However, adding is computationally more efficient, and the known contrast between black and white allows us to take a shortcut

Cell Contrast Analysis – Layer 2

The first step is to create an average RGB value. We accomplish this by dividing the total red, green, and blue values by the number of pixels in each cell.

```
blockSize    = cellsize * cellsize
averageRed   = totalRed   / blockSize
averageGreen = totalGreen / blockSize
averageBlue  = totalBlue  / blockSize
```

The cell contrast is simply the "darkest" pixel value, as calculated by:

```
contrast = averageRed + averageGreen + averageBlue
```

The resulting value is assigned to the cell, and used later for wall analysis.

```
cell.contrast = contrast
```

At the same time, we track the darkest overall cell value

```
if contrast < darkest then
      darkest = contrast
```

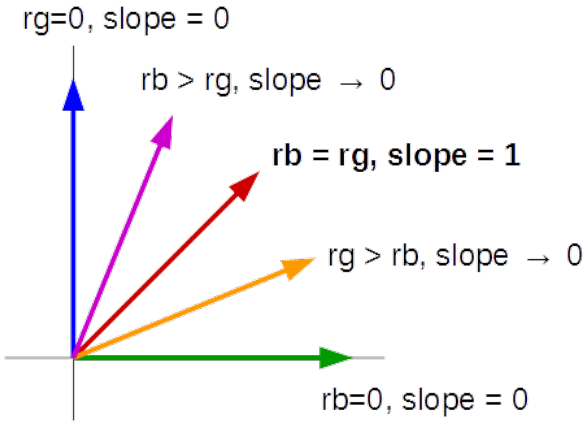
Cell Color Analysis – Layer 2

Red and blue are handled in an identical manner, so we'll look at red as an example for both.

We need to find the “reddest” pixel, so we calculate red's distance from green, and red's distance from blue, for the cell's average color value:

```
rg = averageRed - averageGreen
rb = averageRed – averageBlue
```

Using rg (difference between red and green) as one axis, and rb (difference between red and blue) as the second axis, we are looking for a red value that's equally-distant from both – a line that's extending out at a 45 degree slope.



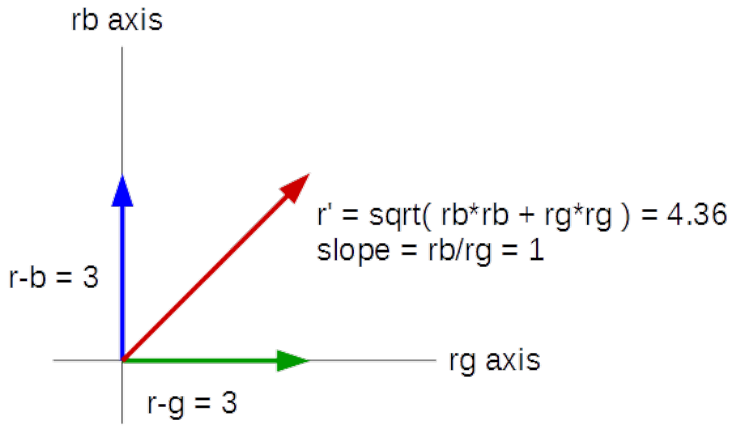
We use the tangent as a fitness function to measure the slope of the line, where an ideal slope (45 degrees) is 1, and anything approaching either axis (0) is increasingly less ideal.

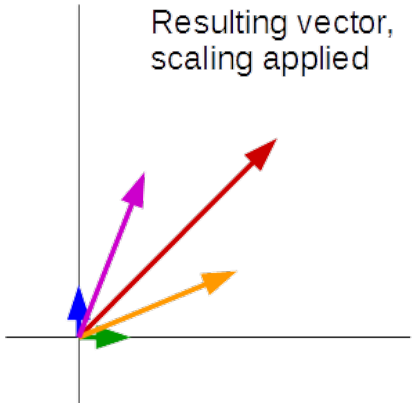
```
If rb > rg then
      slope = rg / rb
else
      slope = rb / rg
```

This returns a value that, as the “red vector's” angle diverges from 0 degrees, returns 0, building toward a return value of 1 as the angle approaches 45 degrees, then decreases from 1 back to 0 as the angle approaches 90 degrees.

We calculate the “red” value using Pythagoras, then scale it by the fitness factor, assuring that the resulting value decreases sharply as it points toward green or blue.

```
finalRed = sqrt(rb*rb + rg*rg) * slope
```





The last step is to track the "reddest" value, and its grid (cell) location:

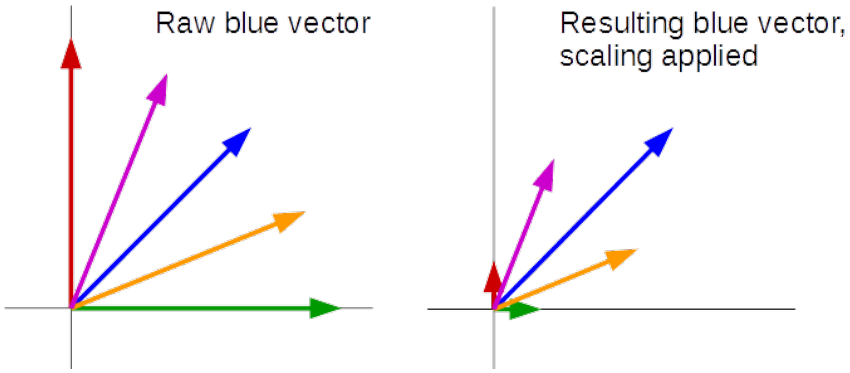
```
if finalRed > reddest then
      reddest   = finalRed
      reddest.X = cell.X
      reddest.Y = cell.Y
```

Further, we only have to perform this calculation if "red" is the largest color value – otherwise, we can simply skip the entire process. Putting all of this together, we have:

```
if (averageRed > averageBlue) and (averageRed > averageGreen) then

      rg = averageRed - averageGreen
      rb = averageRed – averageBlue

      If rb > rg then
            slope = rg / rb
      else
            slope = rb / rg

      finalRed = sqrt(rb*rb + rg*rg) * slope

      if finalRed > reddest then
```

```
reddest   = finalRed
reddest.X = cell.X
reddest.Y = cell.Y
```

The same approach is used for finding the bluest (finish) cell.

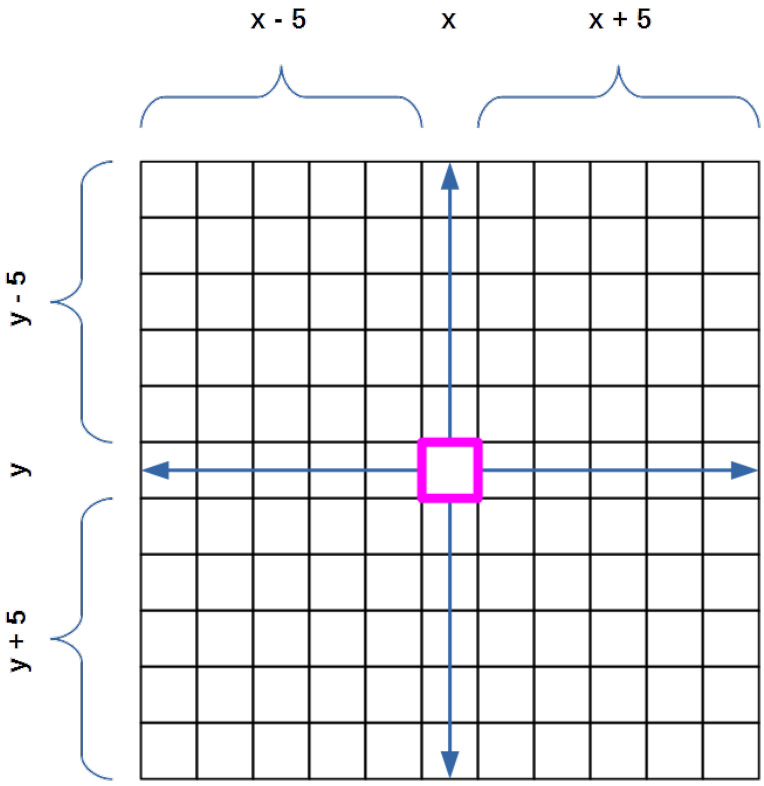


The x,y grid location of the start and end coordinates are explicitly tracked – everything else is dynamic.

Wall Analysis – Layer 3

The wall analysis consists of the average contrast of its local neighborhood, compared to its own contrast.

Through experimentation, the ideal “cell neighborhood” used to find wall contrast was determined to be plus and minus 5 cells of the target cell, in each direction.



Cell Neighborhood for Wall Detection

Wall analysis looks at each cell's contrast value, compared to the average contrast of its “neighborhood”. If the neighborhood's average contrast compared to the cell's contrast exceeds a specific threshold, then the cell itself is a “wall”, otherwise, it's “empty”.

Remembering that the contrast value is the minimum pixel contrast value within the cell, the pixel contrast is calculated as **red + green + blue**.

Recalling that the raw color value is stored as three 8-bit fields, one each for red, green, and blue, this results in a value range for any of the three of **0 to 255**.

This means that there is a maximum contrast (pure white cell) of **3 x 255 = 765**

The minimum cell contrast (pure black) would be **3 x 0 = 0**

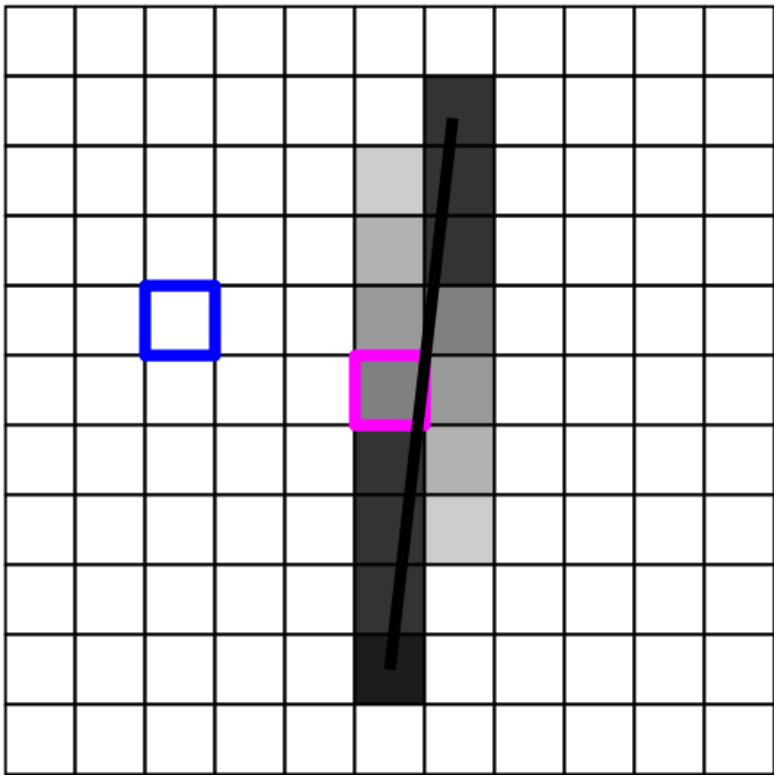
Because we're dealing with an analog source, pure white and pure black are equally unlikely – the "whitest" cell is probably light gray, and the "darkest" cell is probably dark gray, and the overall contrast is a range between the two values.

Because any given maze would be mostly white (black walls drawn on white paper), the average for any given neighborhood is going to be mostly white (or light gray), allowing even a small threshold to detect a "wall" cell.

Experimentally, this threshold was determined to be **20**.

Because the bitmap is a photograph, perhaps taken in less-than-ideal lighting, and because lighting conditions tend to change as a function of position within the bitmap, we can't use the entire bitmap's average to find a specific cell's contrast – for example, if a "white" cell in the lower-left is poorly-lit, it might appear as gray. Meanwhile, if a "black" cell in the upper-right of the same bitmap is over-exposed, it might appear to be the same shade of gray.

The local contrast approach provides a large sampling of nearby cells (which we assume are mostly white), yet adapts well to gradually-changing lighting conditions.

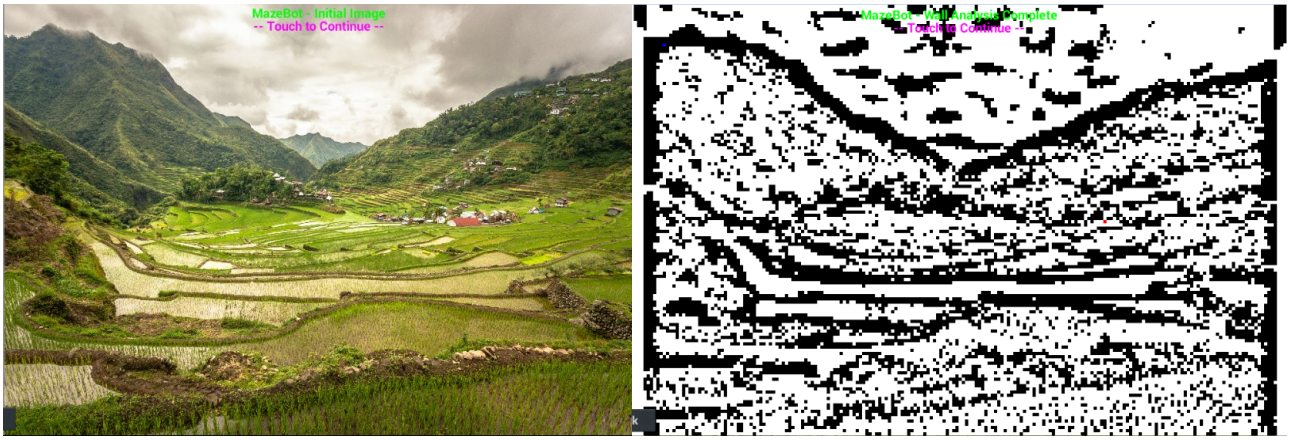


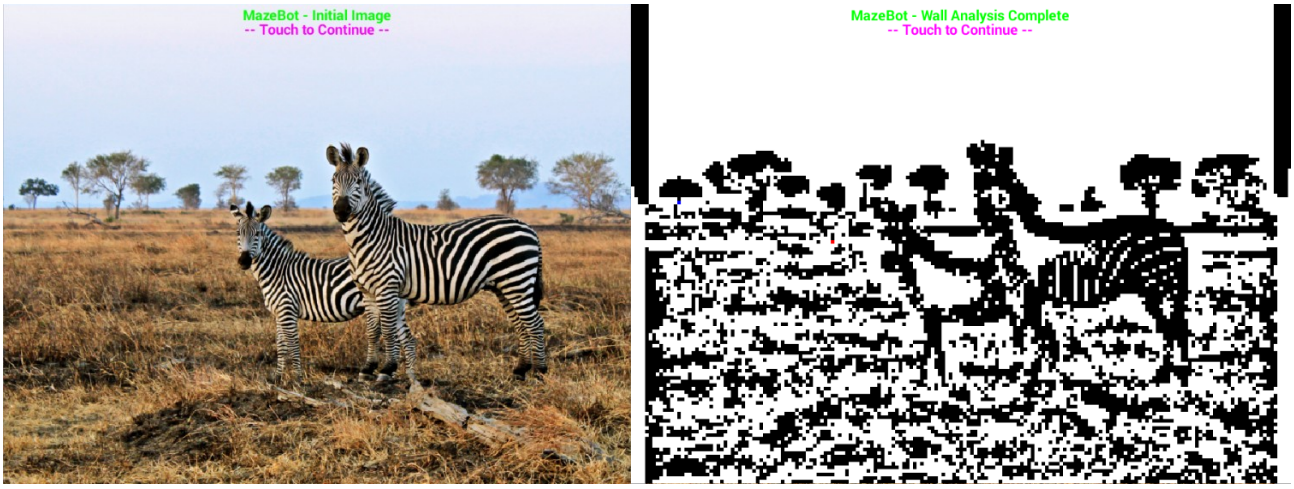
In the figure above, we see a cell within it's 11 x 11 neighborhood. The cell appears to be part of a wall, but because of the "aliasing" effect created during the analysis phase, it's difficult to tell. If the threshold is too high, fewer cells become walls, which can result in gaps. If the threshold is too low, walls can become overly-thick, and can even become conjoined.

Meanwhile, the cell outlined in blue, even though it has *some* dark neighbors, clearly aligns with the majority of non-wall cells, falls within the threshold, and is therefore “empty”.

One byproduct of the “contrast” approach is that both red and blue cells appear black – we'll deal with this as part of the state machine logic.

As it turns out, this approach is extremely fast and accurate at edge detection in general. Here are some bitmaps, scaled to device resolution, that have been processed by the wall-detect algorithm:





Overlay Start and Finish – Layer 3

The final task is to set the start cell's state, and finish cell's state.

There is nothing fancy here – we rely on the x,y grid location that we captured during Layer 1 for each of these, and simply, explicitly set the appropriate cells' states respectively.

As mentioned previously, the wall detect algorithm results in a “start blob” and a “finish blob” - we only accurately find the “reddest” and “bluest” (each) cells, which means that the start and end location could simply be trapped – surrounded by “wall” cells that are not quite as red or blue, at the start and end of the maze, respectively.

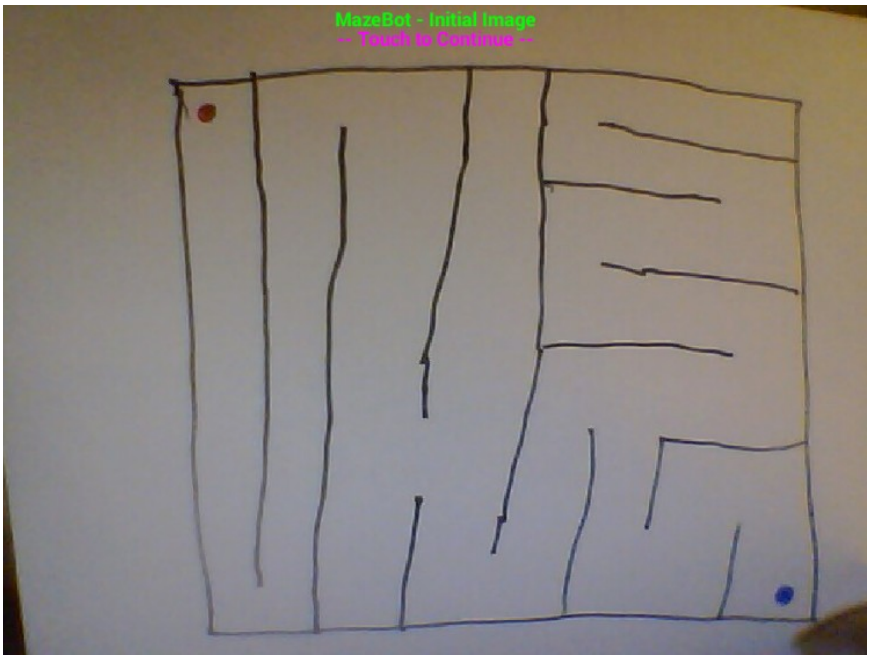
Initially, attempts to resolve this problem by adjusting the wall threshold and color detection process were largely unsuccessful. Instead, later, we'll use a special machine state to resolve this problem.

Sample Processed Maze Image

At the end of layer 3, we're done with image and wall analysis, and we're ready to proceed with the state machine solver.

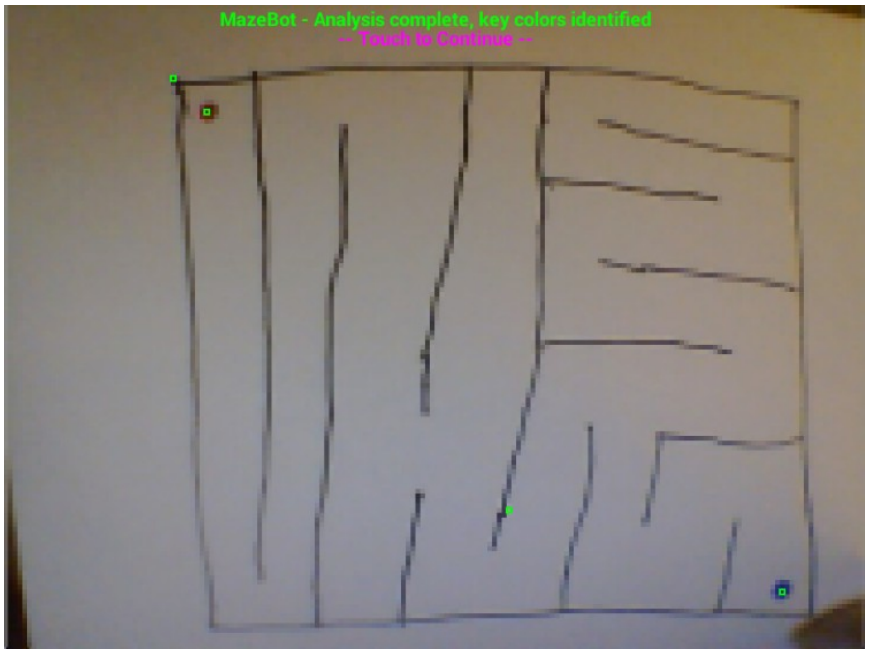
Let's take a minute to walk through the image analysis process, and its artifacts.

Let's start with a sample hand-drawn maze:



We see that lighting quality is poor, resulting in over-exposure in the bottom left and upper-right corners, discoloration, some portions are out of focus, and there are plenty of “JPEG artifacts” bordering many of the walls.

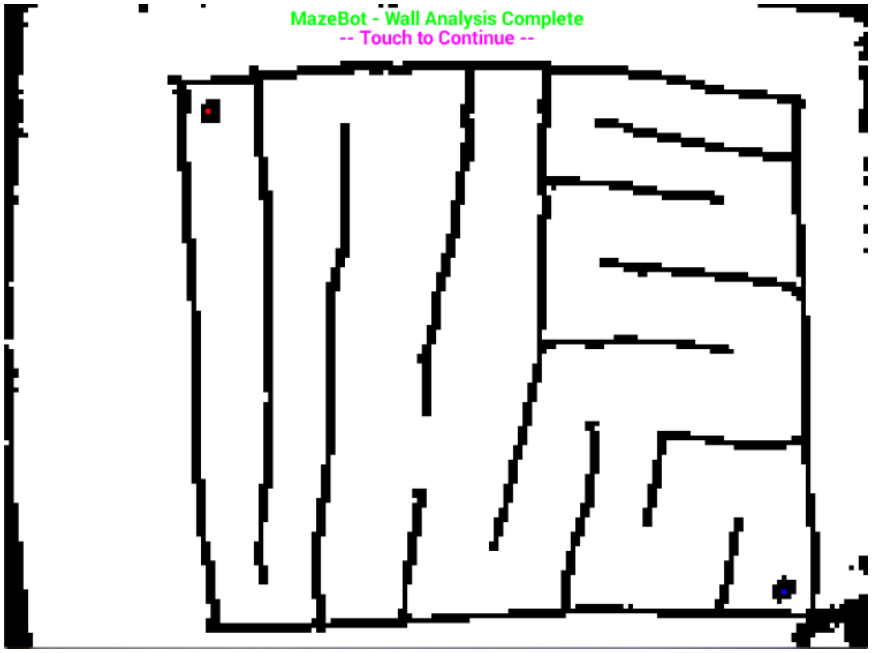
Here is the resulting image after being processed by layer 2:



We see that “wall”, “start”, “finish”, and “empty” cell types have been correctly identified, despite the

fact that some wall elements, due to JPEG artifacts, almost look blue. We also see “aliasing” artifacts for many of the walls – if our layer 3 threshold is too high, there will be gaps in walls. If it's too low the walls will be too thick, and the “start” and “finish” blobs will merge with the adjacent walls.

Here is the same image after layer 3's wall detection process:



Every wall has been identified, with a thickness of at least one cell. Empty passageways are clearly identified and passable. As discussed, we have the “start” and “finish” blob problem.

“Your mileage may vary” - this is an analog image that's converted to a discreet bitmap, and then munged in order to obtain the algorithm's “best guess” about what the user intended, with respect to start, finish, and walls. Some image tweaking may be necessary, and over time, the user will gain experience regarding techniques that translate well, and those that don't, and will be able to adjust their “style” accordingly – how the maze is drawn, lighting conditions, camera distance, angle, etc...

Key Parameters

At this point, key parameters are hard-coded, and are set to experimentally-determined optimal values. Later versions of the program may allow the user to change these:

Parameter	Value	Impact
Vertical cell count	120	This value directly impacts the **cellsize**, or the number of pixels per cell, which is computed by dividing the display height by this value. Larger values result in more, smaller cells. This accommodates finer detail, but requires more computing power per cycle, and thus processes more slowly. Smaller values result in fewer, larger cells. Large cells can fail to detect narrow passages, but are very compute-friendly (much faster). Any increase (y) in cell count (x) results in 2xy + y^2 additional cells to compute for each cycle. As conventional wisdom dictates, as y approaches x, this represents a doubling of both the vertical and horizontal cell resolution, resulting in 4 times the original resolution, and 300% net additional cells that must be computed: If x = y: vertical: x + y = x + x = 2x horizontal: x + y = x + x = 2x resolution: 2x * 2x = 4x^2 increase: (4x^2 – x^2) / x^2 = 3x^2 / x^2 = 3 (or, 300%)
Local neighborhood	5	Used during wall detection, for a given cell at x,y, cells in the range of (x-5..x+5, y-5..y+5) will be used to determine average local contrast. A value of 5 results in an 11 x 11 grid of cells – {x-5..x-1, x, x+1..x+5} (11 elements) by {y-5..y-1, y, y+1..y+5} (11 elements). Because we ignore the cell value itself, the number of cells in this neighborhood is: (2n + 1)^2 - 1 For a neighborhood size of 5, this results in a local neighborhood of 120 cells. A larger local neighborhood means that wall detection is more crisp and accurate under ideal conditions, but runs the risk that gradual changes in the image (due to artifacts of the acquisition process) will result in

		incomplete wall detection. A larger local neighborhood also means that the number of computing cycles for wall detection increases, even though this is largely mitigated by the fact that wall detection is completed in a single pass. A smaller neighborhood runs the risk of detecting too many walls, as insufficient cells are analyzed when making the determination. Increasing n-1 to n (a net increased neighborhood size of 1) results in: each cell: 8n + 8 additional neighbors entire grid: gridWidth * gridHeight * (8n + 8) A net increase of m, where: n' = n + m results in an additional neighborhood cell count of: 8 (n' (n' +1) / 2 - n (n + 1) / 2) For example, if the n = 3, and we increase by 2: original neighborhood: (3*2 + 1)^2 -1 = 7^2 -1 = 48 new neighborhood size: n' = n + 2 = 3+2 = 5 new neighborhood: 8 ((5 (5+1)/2) - (3(3+1)/2)) 8 ((5 * 6 / 2) – (3 * 4 / 2)) 8 (15 – 6) 8 * 9 72 Size 5 neighborhood: (5*2 + 1)^2 – 1 = 11^2 – 1 = 120 comparing size 5 to size 3: 120 – 48 = 72 As cell size decreases, a larger local neighborhood is required in order to maintain a proper view of “local contrast”. Because cell size is determined by the Vertical Cell Count (see above), which is initially-fixed at 120, there probably needs to be a fixed ratio of 120:5 (or close) between the vertical cell count and the local neighborhood size, although this hasn't been experimentally verified.
Wall threshold	20	If the average contrast of a cell's neighborhood (excluding itself), subtracted from the cell's contrast is greater than this threshold, then the cell is a “wall”, otherwise, it's “empty”.

		wall = avgNeighborContrast – cell.contrast > threshold empty = avgNeighborContrast – cell.contrast <= threshold Although changing this parameter has no impact on computing requirements, it affects how accurately walls are interpreted. Increasing this value means that cells with a middle gray value might be ignored, resulting in gaps where a wall should exist. Decreasing this value means that cells with a light gray value might inadvertently be interpreted as a wall, and can result in conjoined walls and other artifacts.

Application (Machine) States

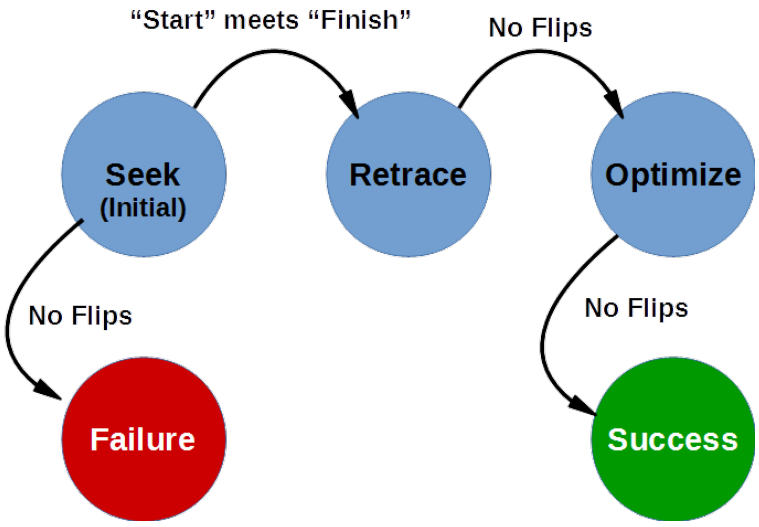
The solver algorithm uses four main machine states to solve the maze and then optimize the final path. A fifth machine state must be employed in order to overcome the “start” and “end” blob problem.

Each pass of the cell grid counts the number of cell transitions from one state to another, known as “flips”. If the flip count is zero after a pass of the grid, then the cell logic has reached a stable state, and the machine logic reacts differently based on its current state.

State	Description
99	**Fix start / end blobs** “Wall” cells that are adjacent to “start” or “finish” are converted to “start” and “finish”, respectively. This effectively “eats” the start and end blobs, but if one of these is conjoined to a wall, the algorithm will start to eat the walls of the maze, considering any connected wall to be part of the solution set. Other than acting on walls rather than empty space, this is identical to the “seek” state (state 1) Each converted cell stores a distance metric, which is the max of its neighbors' distance metrics, plus 1. So if the cell's highest-metric neighbor is 20, the cell itself will be metric 21. Any cells converted by this cell will be 22, etc... When the cell state logic is stable (zero flips), the machine switches to “seek” state (state 1).

1	**Seek (Flood fill)** “Empty” cells that are adjacent to “start” or “finish” are converted to “start” and “finish” respectively. This approach is effectively a flood fill, where each successive generation progresses in all possible directions, by the distance of 1 cell. Each converted cell stores a distance metric, which is the max of its neighbors' distance metrics, plus 1. So if the cell's highest-metric neighbor is 20, the cell itself will be metric 21. Any cells converted by this cell will be 22, etc... When a “start” cell has at least one finish neighbor, a path through the maze has been found, and the machine switches to “retrace” state (state 2) If the cell state logic is stable (no flips), then no solution to the maze is possible – every cell has been flooded without connecting start to finish. In this case, the machine terminates (state -1)
2	**Retrace** In the retrace state, the cell logic works backward from the join cell to both the start and end points simultaneously, following only the most efficient path based on decreasing cell metric. The resulting solution set includes all possible solutions of equal length – we'll explore this in more detail while examining the cell state logic. Once the cell state logic becomes stable, the machine switches to the vertical optimizer (state 3)
3	**Vertical optimizer** As stated, the resulting solution set includes multiple possible pathways. The vertical optimizer reduces vertical redundancy, resulting in a solution set whose horizontal pathways are only 1 cell wide. When the cell state logic becomes stable, the machine switches to the horizontal optimizer (state 4)
4	**Horizontal optimizer** The horizontal optimizer reduces horizontal redundancy, resulting in a solution set whose vertical pathways are only 1 cell wide. Running vertical and horizontal optimization concurrently, results in shearing, and creates gaps in the resulting pathway. When the cell state logic becomes stable, the machine switches to the “terminate – success” state (state -2)

-1	Terminate – Fail
-2	Terminate – Success



Cell State Logic

Cell state logic rules change based on the overall machine state. We will look at each machine state and its associated cell state logic in detail.

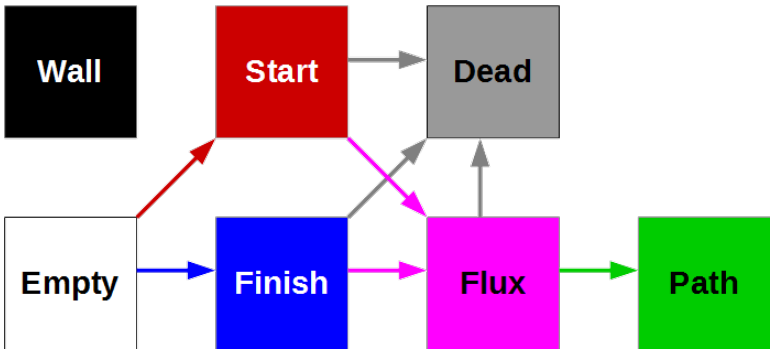
Overview of Cell States

Each cell can exist in one of multiple states:

Cell State	Description
0 - Empty	Empty space represents any viable pathway, and may change to another state as the solution progresses.
1 - Wall	Walls are generally immutable. They are generated as a result of contrast and wall analysis, and remain static through the course of the solution. Walls are obstacles that define the parameters of the resulting solution.
2 - Start	One start cell initially marks the start location. As the flood fill (seek) machine state progresses, empty neighboring cells are converted to "start" cells.
3 - Finish	Like start, there is initially 1 finish cell, and as the flood fill (seek) machine state

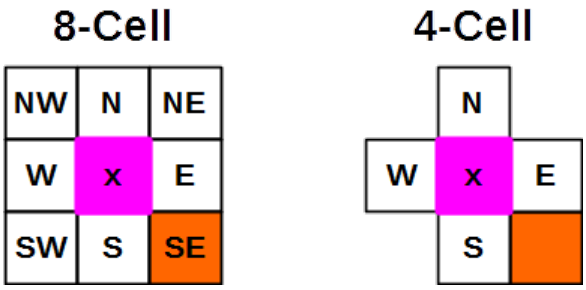
	progresses, empty neighboring cells are converted to “finish” cells.
4 - Flux	Cells switch from start or finish to “flux” state during the retrace process, and at the end of the retrace machine state, the set of all flux cells represents the set of all possible equidistant solutions.
10 - Path	During optimization, the final result set of “flux” cells are converted to “path” cells.
5 - Dead	Cells that are not part of the final path are systematically killed off, and end up in the “dead” state.

Here is a cell state transition diagram:



Cell Neighborhood

Although some specialized cell neighborhood configurations exist, most cellular automata either use an 8-cell neighborhood, consisting of the cardinal directions, plus diagonals, or a 4-cell neighborhood, consisting of ONLY the cardinal directions.



In the figure above, we see that in the 8-cell neighborhood, the magenta cell in the center has 8 neighbors, and the orange cell is a direct neighbor (one “hop” away).

We also see that a cell with 4 neighbors doesn't include the cell to its diagonal, and the same orange cell is two “hops” away – it's now a neighbor of one of the magenta cells direct neighbors.

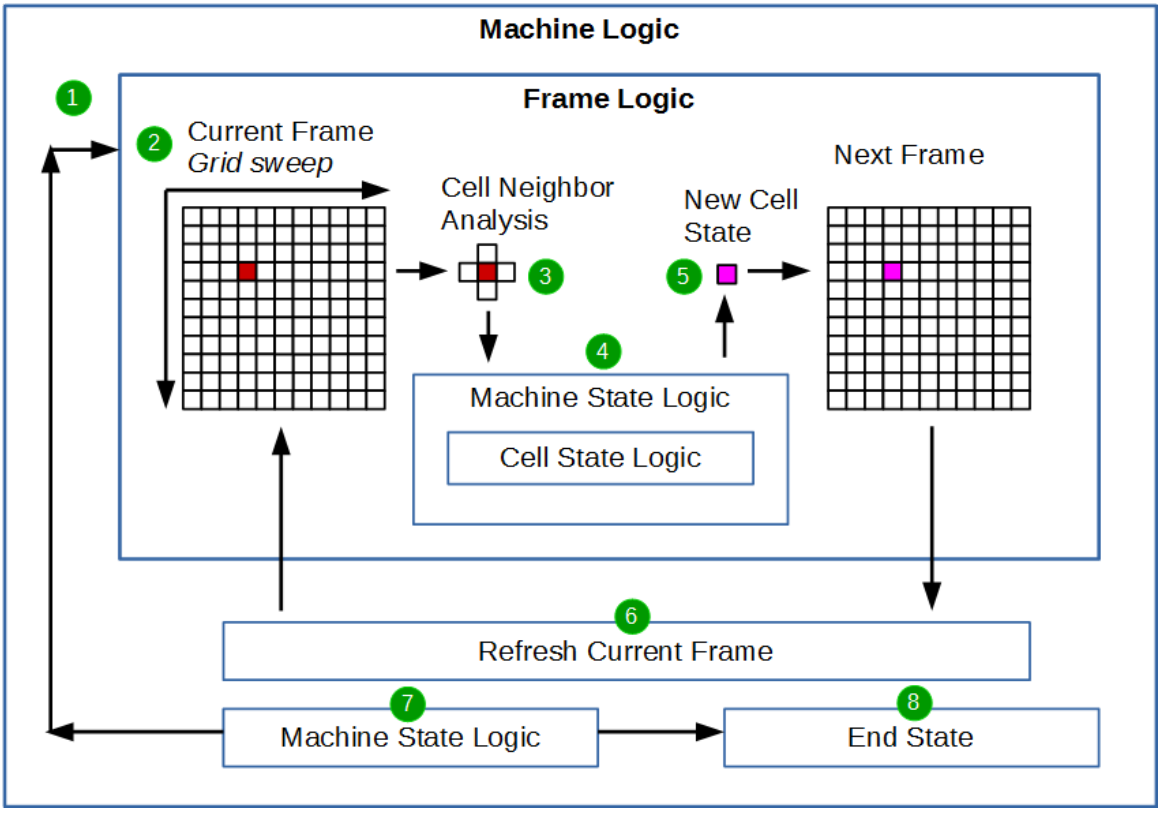
This application uses a 4-cell neighborhood, facilitating a final result set that can be interpreted as left-right turns.

Cell Structure

Each cell contains the following attributes:

Cell Attribute	Description
State	The current cell state
Metric	An aggregate cost from start or finish, used to determine optimum and sub-optimum paths

Application Flow



1. When the solver executes, program flow immediately moves from the outer "machine" logic to the "inner" frame logic.
2. The frame logic performs a grid sweep, examining each cell. The term "frame" refers to the entire grid in its current state. A clean "next frame" grid is used as a temporary storage location to store any changes to the current grid state. We will examine the reasons for this approach later.
3. Neighbor analysis is performed for each cell – this consists of a tally of the number of neighbor cells of every state, plus its neighbors' maximum and minimum metric values.
4. Machine state logic dictates which cell state rules are followed. At a minimum, the original cell state is copied to the new frame. Cell state logic dictates the new cell state, and any changes to the metric value.
   Machine and Cell state logic determine specific code blocks to execute, that may change the cell's state or other attributes, perform checks, perform calculations, etc...
5. The new cell is copied in to the "next frame".

6. Once the grid sweep is complete, a refresh sweep updates the current frame using state and metric information stored in “next frame”. Along the way, it counts the number of cell state transitions, called “flips”.
7. Zero flips indicates that cell state logic has stabilized. Machine state logic looks for stable cell states, along with other logic, to determine if the machine state needs to change.
8. If the machine logic determines that no further processing is required, it moves in to an “End” state, where the output layer takes over. (See “Solution Overview”, above)

The reason we use a “next frame” buffer to store any cell updates, rather than make updates directly to the current frame, is that any changes to the current frame during the analysis process causes bias and corruption.

For example, let's say that a given cell's state changes. This will affect its neighbor's behavior *during the grid sweep*, resulting in a bias toward the original cell's new state.

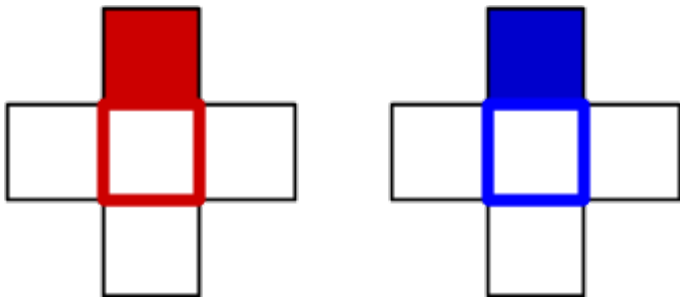
This effect leads to corruption of the entire grid.

By using a “frame buffer” to temporarily store changes to the current frame, we keep the current frame pristine during the grid sweep analysis, and we create a clean “next frame” that consists of a copy of the current frame, plus any updates that have been applied.

The machine logic then performs a quick sweep to copy the “next frame” cell states to the current frame, and count cell state transitions (flips).

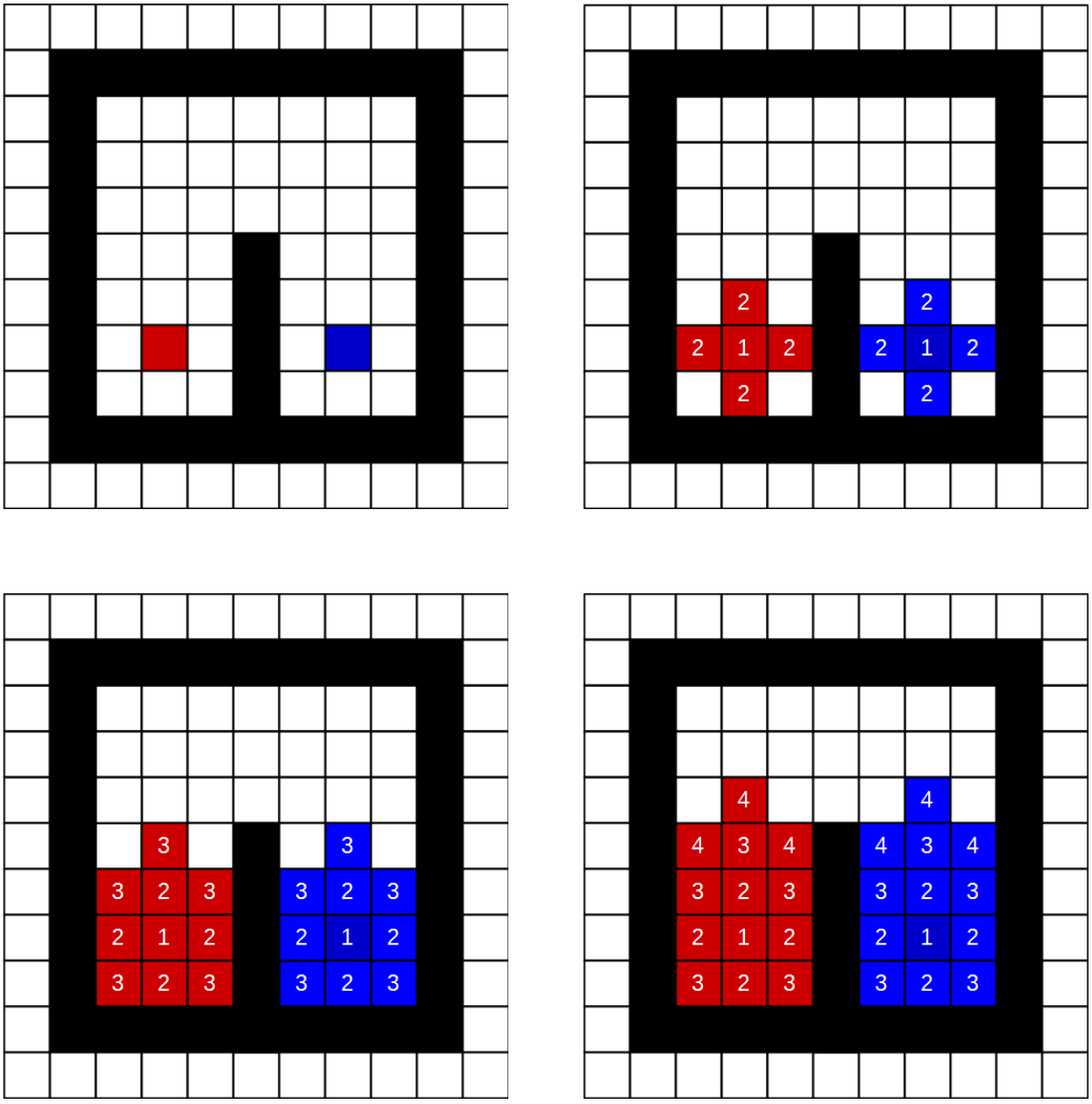
Machine State “Seek” (1) Cell Logic

In the “seek” state, empty cells that directly border either start or finish cells are converted accordingly.



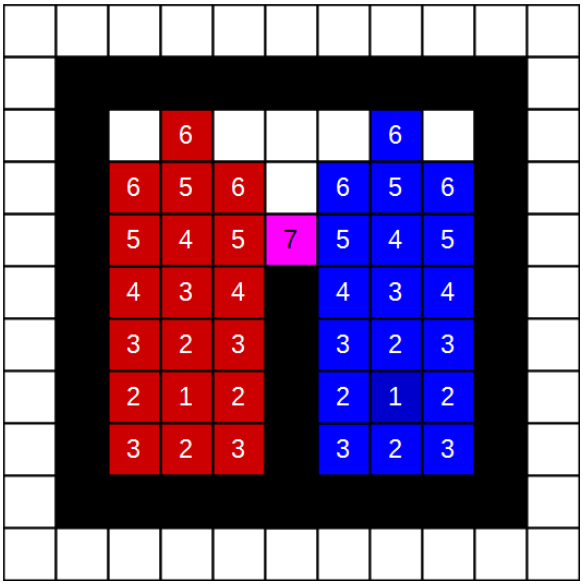
This results in a “flood fill” of both start cells and finish cells, that gradually works toward some common connecting cell.

As each cell is converted, it takes on the metric value of its largest neighbor, plus 1.



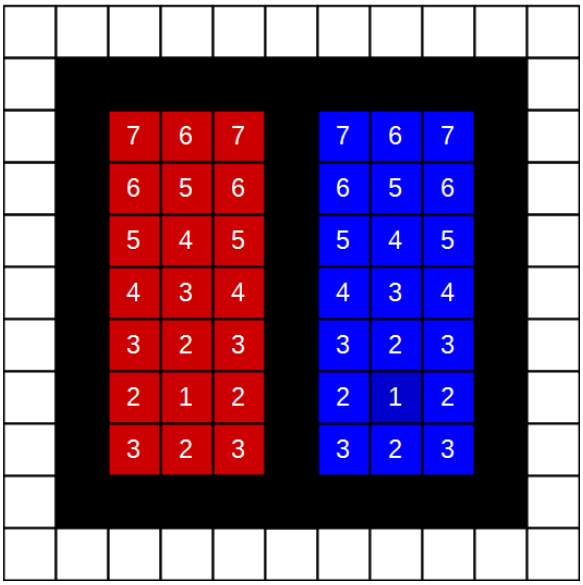
In the figures above, we see the flood fill effect progress simultaneously from both the start and finish locations. We also see the resulting cell metrics.

Eventually, there will be a "start" cell with at least one "finish" neighbor – this is known as the "join" cell, which is thrown in to a "flux" state, and the ***machine state*** is changed to state 2 (Retrace).



In the figure above, we see that we have found the join cell, after a couple more frames have gone by.

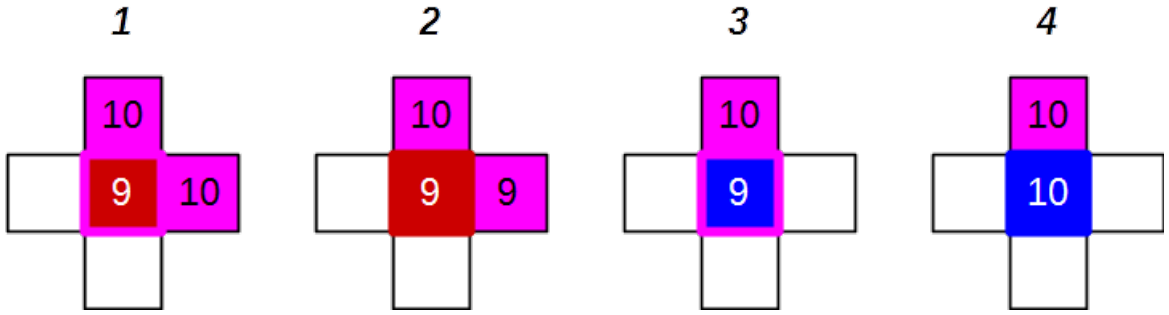
If the maze isn't solvable, as in the example below, the machine state logic kicks in, looking for stable cell logic (no cell flips have occurred in the previous frame), and the ***machine*** switches to a failed end state.



(Example where no solution is possible)

Machine State "Retrace" (2) Cell Logic

The "retrace" state attempts to find the most efficient path through the start and finish cells by using the cell metric values.

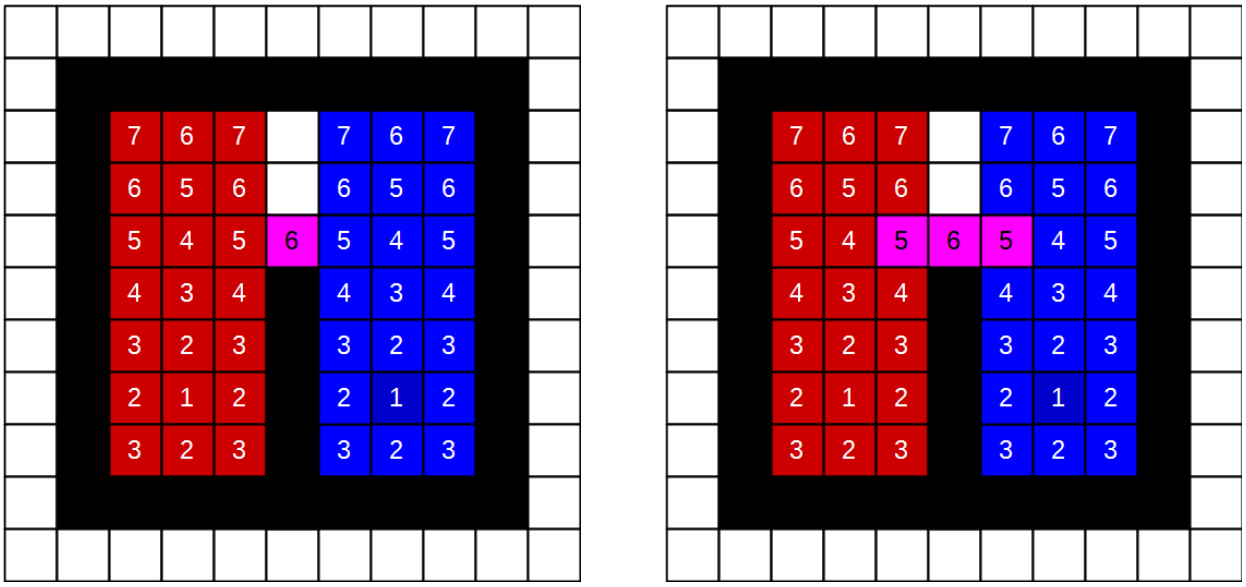


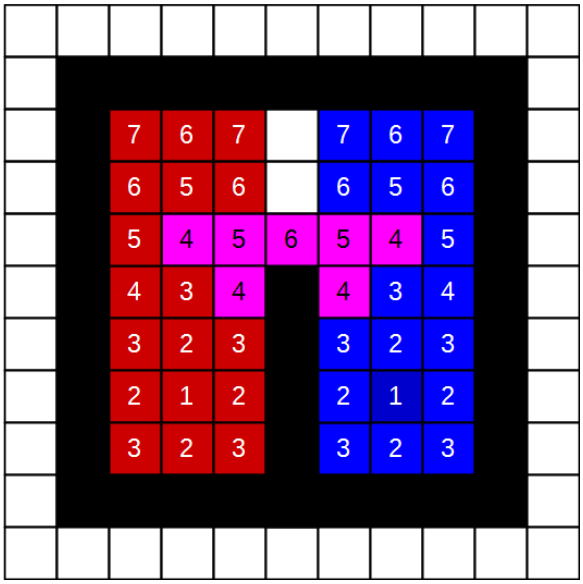
If a start or finish cell borders a "flux" cell, ***and*** it has a lower metric than all of its "flux" neighbors, then its new state will be "flux".

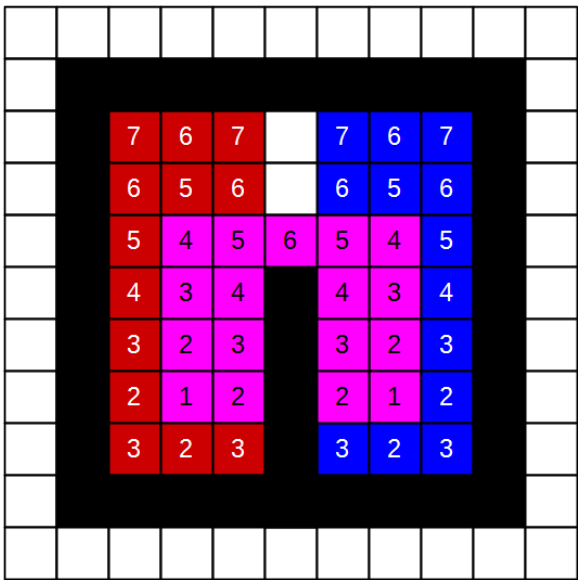
If the cell has a "flux" neighbor with the same or higher metric, it will keep its current state.

In the figure above, we see four examples. In example 1, the "start" (red) cell switches states to "flux" (magenta), because it borders a "flux" cell, ***and*** its metric is lower than all of its flux neighbors. In example 2, the red cell maintains its state because, although it borders a flux cell, there is already a flux cell with the same metric (thus, the cell in question is *not* on the least-cost path). In example 3, we have a "finish" (blue) cell with one flux neighbor, and it changes states because its metric is lower. In example 4, the finish cell maintains state because its metric is the same as its flux neighbor.

Starting with our simple example from machine state 1, we can watch the retrace process continue:







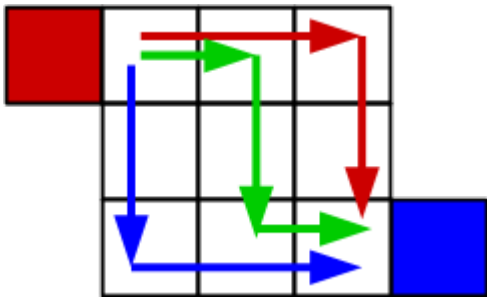
In the figures above, we start with the join cell, as we transition from ***machine state*** 1 to 2. The "flux" state progresses back to start and finish simultaneously, following only the least-cost path, and ignoring higher-cost cells.

Eventually, in the final frame, we see that the cell state logic is stable (no flips), because all neighboring start and finish cells have a higher or equal metric, and are thus ignored.

Once the cell state logic is stable, the ***machine*** switches to state 3 (vertical optimizer).

Machine State "Optimizers" (3 and 4) Cell Logic

After retrace, the resulting set of flux cells represent the set of all **equal-cost solutions**, and thus optimization is necessary, to pick a single path.



For example, think of a car that can only make left or right turns, taking any efficient path from the red square to the blue square, in the figure above. Regardless of what route the car takes, it must cross exactly 5 white squares.

Likewise, the red path, green path, and blue path are all of equal length (and there are many more possible paths).

The purpose of the vertical (machine state 3) and horizontal (machine state 4) optimizers is to eliminate duplication, resulting in a one-cell-width path from start to end.

Each requires a discreet machine state, because they can't run concurrently – running both optimizers at the same time results in shearing, which creates gaps in the final pathway.

Therefore, from a machine state perspective, we perform all possible vertical optimizations, look for a stable cell state, and then switch to the horizontal optimizer.

Once the horizontal optimizer completes (stable cell logic), the ***machine state*** switches to "end (success)"

Part 1 – Eliminate unused Start and Finish cells

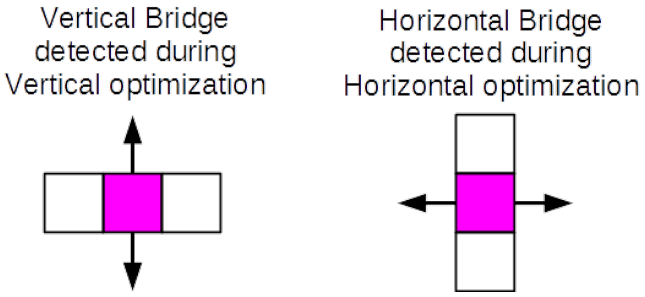
At this point, we know that the set of flux cells contains multiple possible solutions, and we no longer need any remaining "start" or "finish" cells (states 2 and 3, respectively), so we set them to state 5 ("dead")

Since vertical and horizontal optimization occur separately, we only need to perform this part once – during vertical optimization. There's no need to run it again during horizontal optimization.

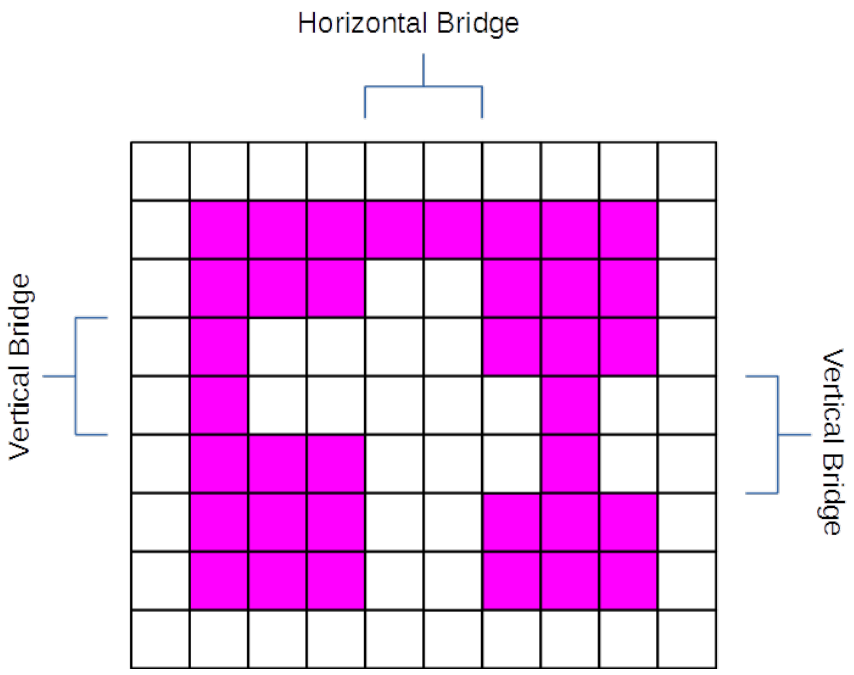
Part 2 – Bridge Logic

In the vertical optimizer, a "bridge" is a cell that has nothing to the left or right of it, thus it's connecting one or more cells above, to one or more cells below, acting as a vertical "bridge".

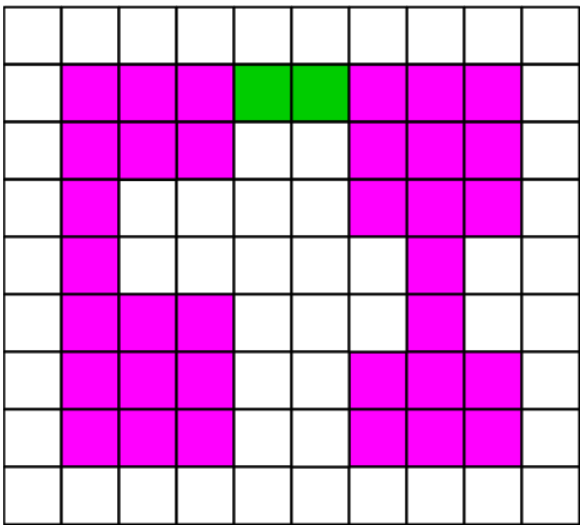
Likewise, the horizontal optimizer detects horizontal bridges, with nothing above or below, and thus the cell connects one or more cells to its left to one or more cells to its right.



The figure below specifies some examples of bridges:



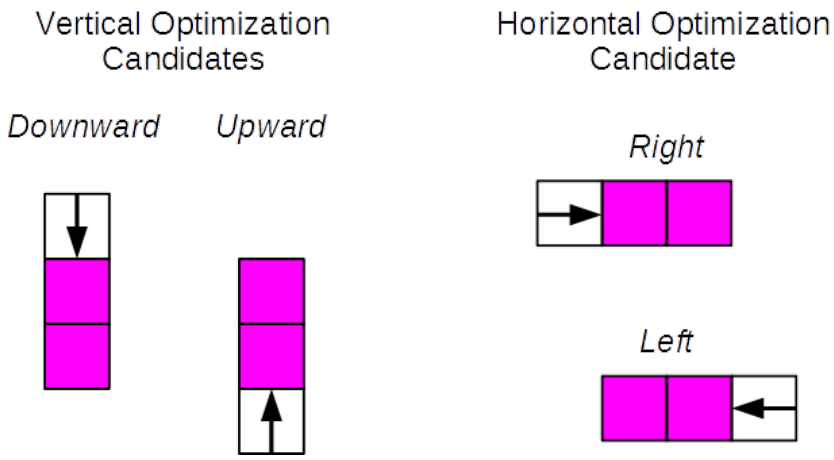
Anything detected as a “bridge” is set to “path” state:



Part 3 – Detect Shearing

Before we discuss shearing, let's discuss the optimization process.

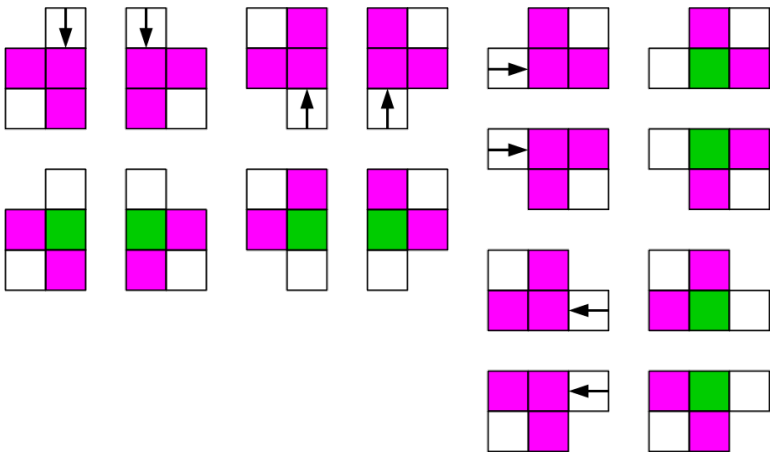
Because the purpose of optimization is to eliminate redundancy, we look for a cell that borders an empty cell on one side, and another flux cell on the other side. We're looking for two adjacent flux cells, and *possible* optimization can occur from the direction of the empty cell, facing a stack of two or more flux cells.



Candidates that might be able to be optimized include the following:

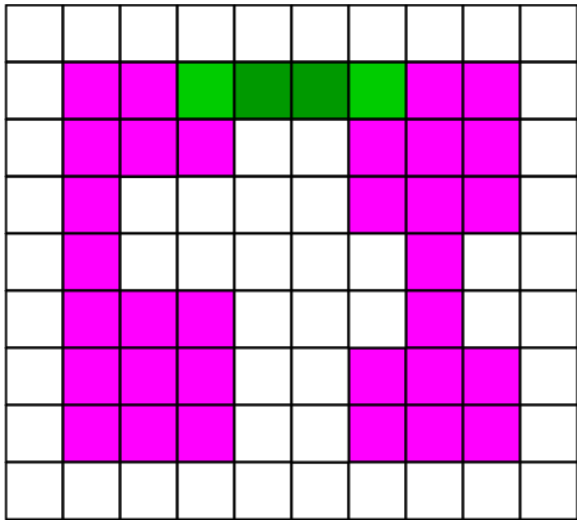
- If a cell has air above, but **another cell below**, it is a candidate for **downward** optimization
- If a cell has air below, but **another cell above**, it is a candidate for **upward** optimization
- If a cell has air to the left, but **another cell to the right**, it is a candidate for **right** optimization
- If a cell has air to the right, but **another cell to the left**, it is a candidate for **left** optimizations

Once we've identified that a cell *could* be optimized, we need to check for shearing, which is the condition where corners get “sheared” off.



The figure above demonstrates all eight possible shearing conditions.

If shearing is detected, as the figure indicates, a corner cell is immediately converted to “path” state to avoid over-optimization.

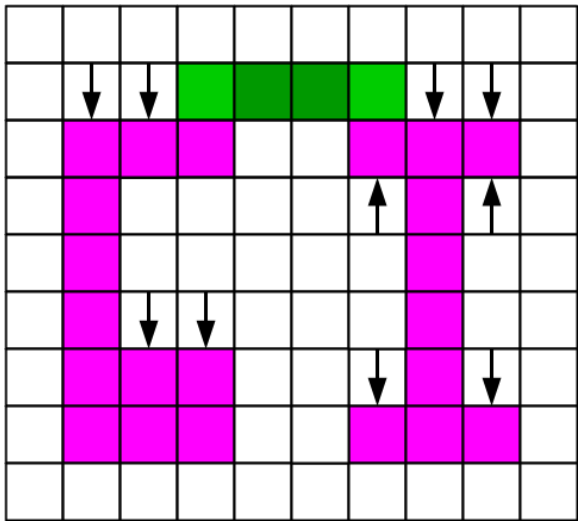


In our example from bridge detection above, shearing detection finds two more cells.

Part 4 – Optimize

Once we know that a cell is not a bridge, and not a shear point, we can try to optimize it.

The last step is to compare its metric with its neighbor – the less expensive of the two survives.



The figure above is the result of one frame of vertical optimization. We clearly see that more bridges and corners have formed. Obviously, there will be some artifacts on the right-hand side, due to the fact that this is an example chosen for clarity, and not a realistic solution set. The example above will result in "serif" nodes in the upper and lower right-hand regions that would not normally form from a properly-created solution set.

Part 5 – Convert remaining Flux to Path

The final part of the optimizer converts any remaining "flux" cells that have survived both the vertical and horizontal optimizer, to "path" state.

This final step is very straightforward – for any "flux" cell, simply change state to "path".

State Logic Framework

It's possible to combine the machine and cell state logic in to a single framework.

The normal approach would be to use branching logic for the machine state, with nested branching logic for each applicable cell state:

```
select machineState
      case 1
            '* Machine State 1 *
            select CellState
                  case 0
                  '* Cell State 0
                  case 2
                  '* Cell State 2
      case 2
            '* Machine State 2 *
            select CellState
                  case 2
                  '* Cell State 2
                  case 3
                  '* Cell State 3
      (etc...)
```

A better and more flexible approach is to multiply the machine state by some constant *c*, adding the cell state. This effectively gives us a single base *c* value that simultaneously represents the machine and cell states, and provides the opportunity to consolidate the branching logic:

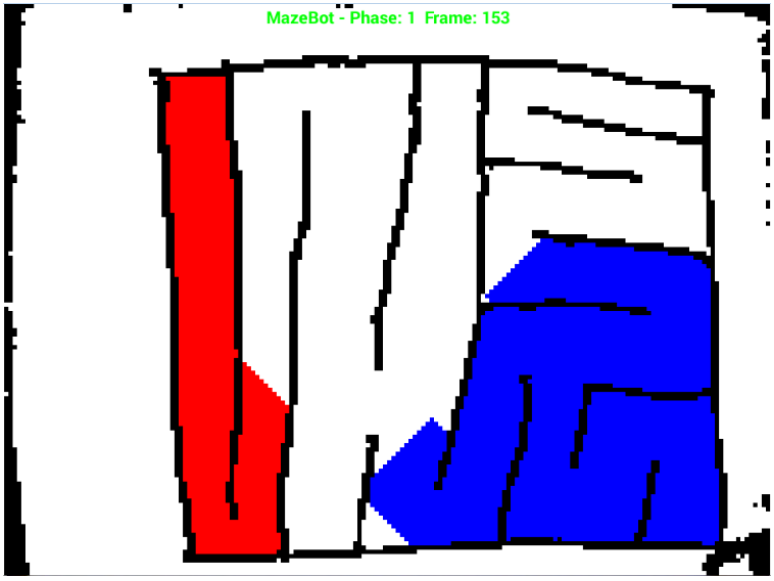
```
c = machineState * 1000 + cellState

select c
      case 1000
      '* Machine State 1, cell state 0
      case 1002
      '* Machine State 1, cell state 2
      case 99001
      '* Machine State 99, cell state 1
      case 2002
      '* Machine state 2, cell state 2
      case 2003
      '* Machine state 2, cell state 3
      (etc...)
```

By taking this approach, the same code blocks can be used across machine and cell states:

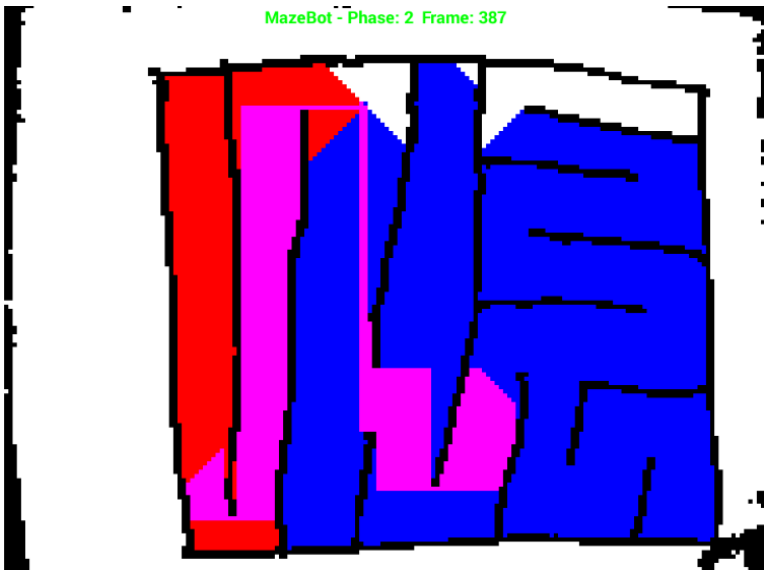
```
select c
      case 1000, 99001
      '* Fill logic *
      case 1002
      '* Detect start / end overlap
      case 2002, 2003
      '* Retrace logic
      (etc...)
```

This approach greatly increases both the overall efficiency, and the flexibility of the design.

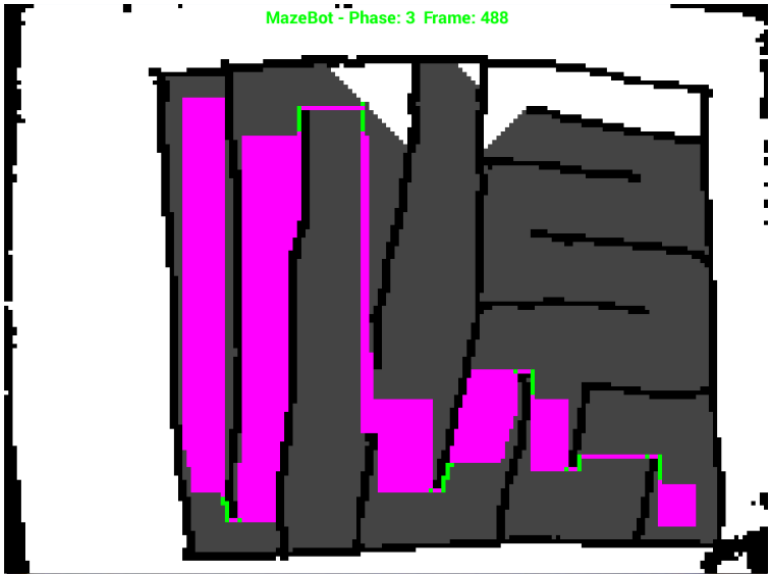
Sample Screens Showing Solver Process



The figure above shows the flood fill (seek) state in progress. Each “frame” expands the red and blue domains by one cell. The diagonal pattern is an artifact of the 4-cell neighborhood, but aids in ensuring that neighboring cells always have a different cost metric, and supports producing a final solution set that could be interpreted in to left-right directions.

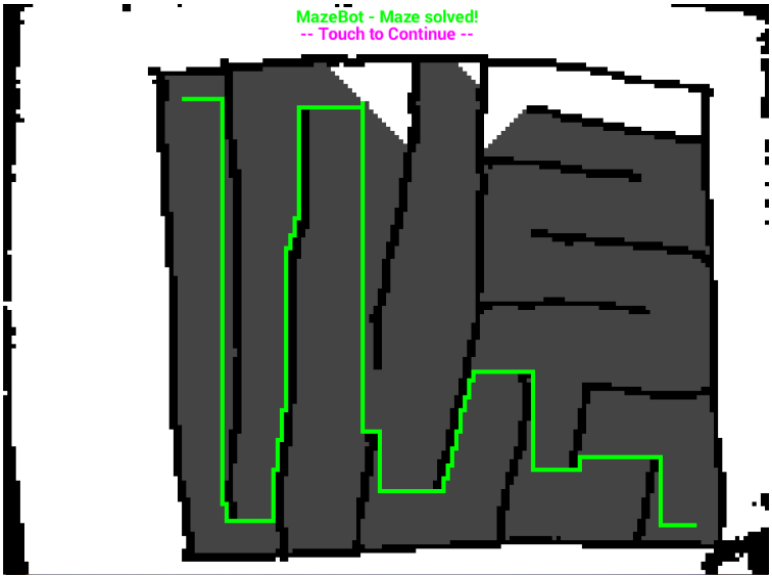


The figure above shows the retrace state in progress. This process identifies a set containing all equivalent solutions.



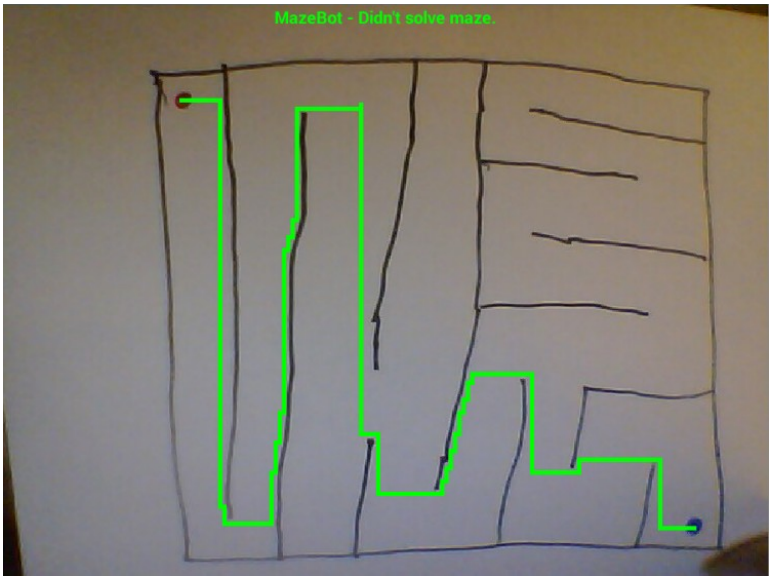
The figure above shows the vertical optimizer running. It has already identified several bridges and corners (possible shearing points), and converted them to “path” status.

Gray cells are dead cells – cells that were “start”, “finish”, or “flux” state, and have been eliminated because they are either not part of the solution set, or they have a more efficient neighbor.



The figure above shows the final solution after optimization. All unnecessary “start”, “finish”, and “flux” cells have been eliminated, and all remaining “flux” cells have been converted to “path”.

Sample Output Layer



The figure above shows the final solution set overlaid on top of the original maze bitmap.

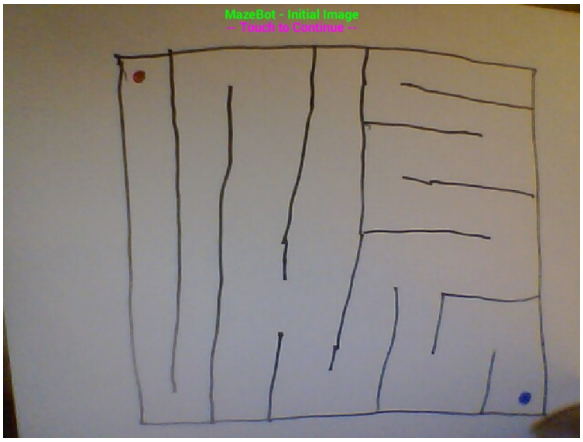
The purpose of the output layer is to produce a graphical or data representation of the solution set – for example, the output layer could be used to produce a text or other data file containing the coordinates for all “path” cells.

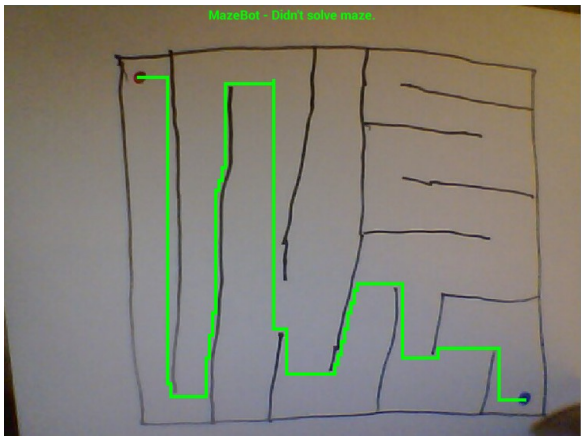
Solution Analysis

In this section, we will look at the project goals, and qualify whether those goals have been met by the solution design.

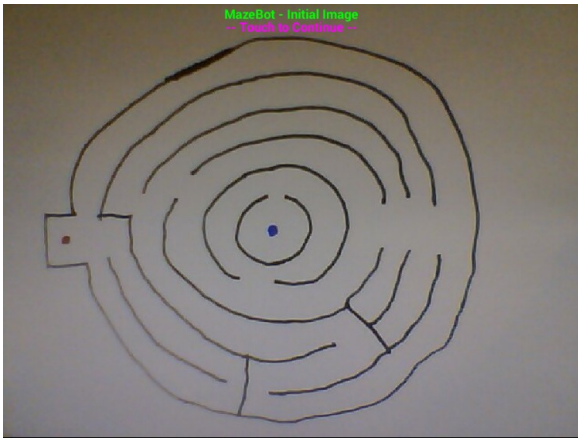
Interpret a Bitmap of a Hand-Drawn Maze

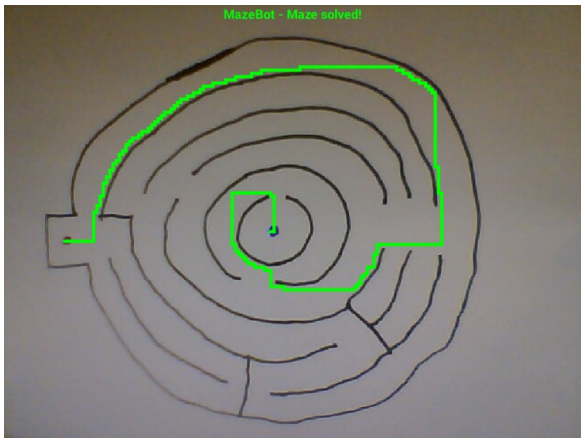
This document has demonstrated both theory and application:



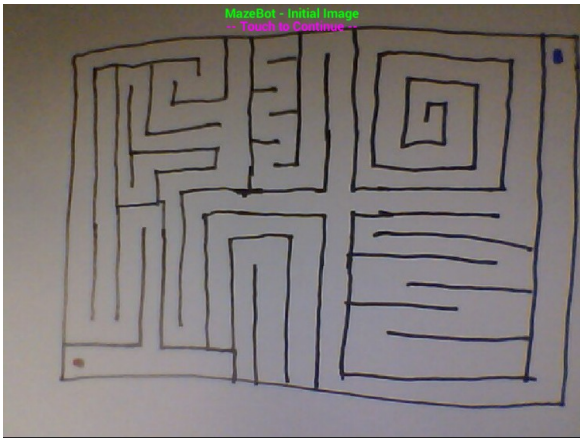


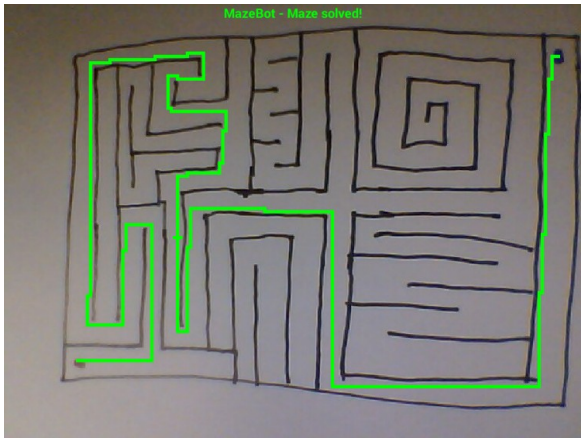
Below, the algorithm is demonstrated, running against a circular maze – typically very difficult for a machine to solve, because the lack of perpendicular walls and well-defined intersections defy node interpretation and analysis. In addition, this maze was specifically constructed to defy being solved by the left-hand rule:





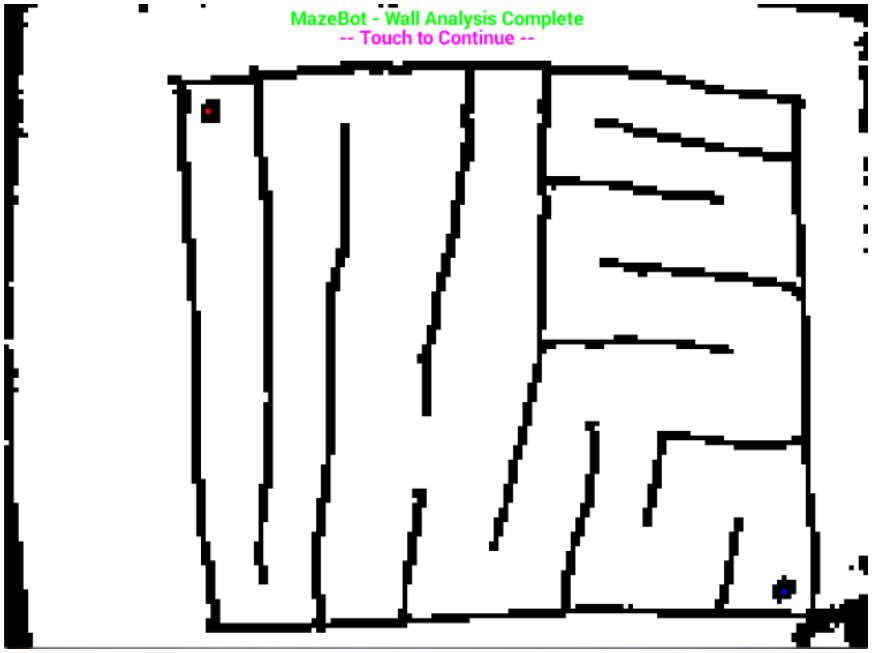
In the following example, we have a more complex maze, with lighting, focus, and color variations:





Convert a Bitmap in to a Cell Array

This document has demonstrated both theory and application



In the image analysis process, we use average pixel values across each cell to find the red and blue start and finish points (respectively).

We find the “reddest” and “bluest” cells by converting color differences in to components of a 2D vector, and then applying a weighting function against the magnitude, based on angular distance from an ideal (equidistant from blue and green in the case of red, or equidistant from red and green in the case of blue).

We use a large “cell neighborhood” to determine local contrast by comparing a cell's minimum contrast value to the average contrast of the cell's neighborhood. If the difference is above a specified threshold, the cell becomes a wall, or empty, otherwise.

Computationally Efficient

We've covered the image analysis process, which has the same fixed computational cost regardless of the image. The first pass aggregates individual pixels in to cells, and the second pass conducts the wall analysis.

Therefore, we'll focus on the machine and cell state logic, as the key driver of computational cost.

As discussed, we can define a “meta state” for each combination of machine state and cell state, by multiplying the machine state by a constant and adding the cell state. This allows for a simplified logic framework, and precludes any looping within the machine / cell state logic code blocks.

For a minimum path length of *n* cells, the following cycles are required:

- n/2 frames in the “flood” state (1). This is because the flood fill occurs from both the start and finish points simultaneously, and both advance by one cell per frame.
- n/2 frames in the “retrace” state (2). This is because the retrace process follows a path from the join cell back to both start and finish simultaneously.
- The number of vertical optimization frames *v* plus the number of horizontal optimization frames *h* can't exceed the path length *n*.
  n>=(h+v)
  The total solution space prior to optimization has the same number of segments and length as the final pathway. For example, if a vertical segment is 20 x 5 cells, it requires no more than 20 cells to optimize. Each additional segment (horizontal or vertical) requires the same.

Therefore, the entire solution requires no more than *3n* frames, where n is the minimum path length.

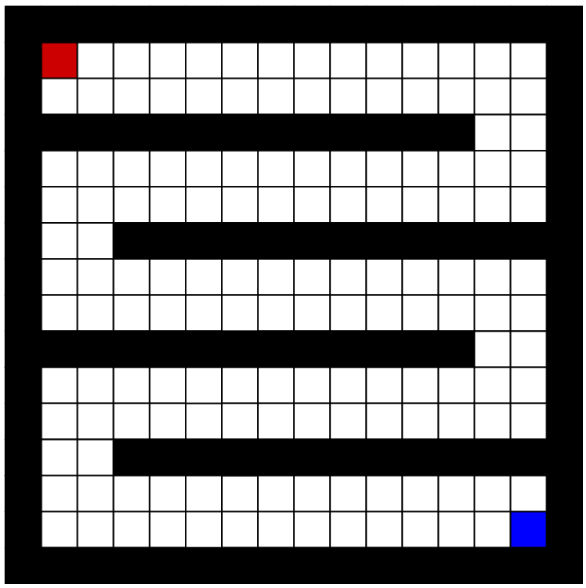
From a complexity standpoint, let's determine the longest path that can be practically formed by an n x n cell array:

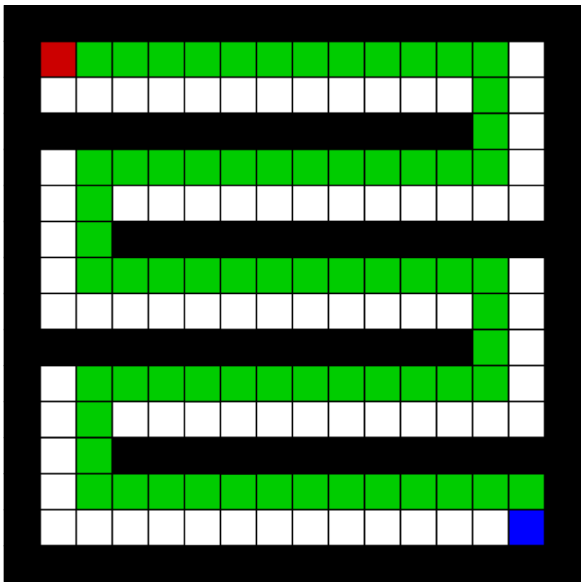
- The entire maze is surrounded by a 1-cell-wide wall
- Wall thickness can be assumed to be 1 cell
- Path thickness can be assumed to be at least 2 cells – if walls and pathways were all 1 cell wide,

other algorithms, such as dead-end-filling CA algorithms, would be more efficient.

- No dead-end paths – the entire maze must be used.
- Start and end points are separated by the longest cell-count distance possible

The resulting shape is a zig-zag. Here is a sample 16x16 "maximally-convoluted" maze and its solution:





The shortest path can be defined as:

- (n-1)/3 total horizontal segments, *h*. This is because each path (of width 2) is bordered by a wall of width 1 (2 + 1 = 3).
  h = (16-1)/3 = 5
- Each *h* segment will have a cell count *hc* of n-3. This is because each vertical path has 2 empty cells, ONE of which must be filled, bordered by two wall cells. 2 + 2 = 4 – 1 = 3.
  hc = 16-4 = 12
- Each *h* segment is connected to the next *h* segment by a *v* segment.
- There are *h*-1 *v* segments. A *v* segment joins adjacent pairs of *h* segments, and there are only *h*-1 adjacent pairs.
  v = *h*-1 = 5-1 = 4
- Each *v* segment cell count vc is the width of the pathway – 2.
  vc = 2
- ONE extraneous cell *ec* must be added to the solution, to accommodate the asymmetric orientation of start and finish.

ec = 1

The final path is therefore:

h * hc + v * vc + ec

Defined in terms of n, the longest path through an n x n maze with wall thickness of 1 and path thickness of 2 is:

(n-1)/3 * (n-4) + ((n-1)/3-1) * 2 + 1
n(n-1)/3 – 4(n-1)/3 + 2(n-1)/3 – 2 + 1
(n-2)(n-1)/3 – 1
(n^2 – 3n + 2) / 3 - 1

In our example above, the minimum path length is:

(16^2 – 3*16 + 2)/3 – 1

(256 – 48 + 2)/3 -1

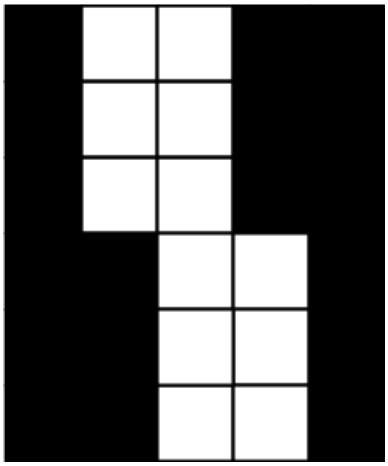
210/3 -1

70 – 1 = **69**

Ignore Shape, Scale, and the Requirement to Define Nodes

As we've seen, pathways are determined dynamically, and the algorithm adapts to any shape or style of maze.

The one requirement is that an empty pathway must be generally 2 cells wide, meaning that any shift still allows a 1-cell-width passageway.



Effectively, the cell size must be 1/2 to 1/3 of the width of a passage in order to have sufficient resolution, to prevent a dead end caused by aliasing.

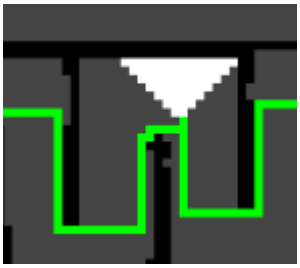
Although the cell size can be reduced, yielding a greater effective resolution, the limit beyond which, this ceases to be beneficial is where the cell size approaches 1 pixel.

Short of this, any bitmap that is properly interpreted with a clear start and finish position, and clear wall and passageway definitions, can be effectively and efficiently solved by the described algorithm.

Produce a Discreet Result Set

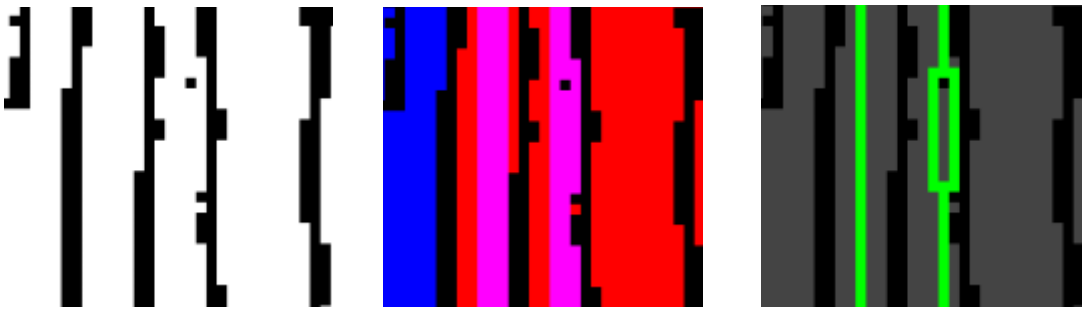
The described solution produces a 1-cell-wide pathway with **two notable exceptions**:

1. **A "knob" attached to the join cell.** This is an artifact of legitimizing a less-optimum path by assigning a high metric to the join cell.



This artifact *could* be eliminated through additional optimization – it's effectively the only "path" cell with one neighbor, other than the true start cell and the true finish cell.

2. **Bifurcation caused by errant, detached wall artifacts.**



In the above example, a wall artifact appears as a random dot in the middle of a pathway. During the retrace stage, we see that the dot is in the middle of the solution set. In the third figure, the final path bifurcates around the artifact.

This can only happen in the rare condition that *both* pathways are of equal cost. To eliminate this artifact, the algorithm should look for a "path" node with three "path" neighbors. Two of the neighbors should be of equal metric, and one should be randomly eliminated.

Using the dead-end fill approach, any cell (short of the true start and end) can then be eliminated, resulting in one of the two paths being slowly killed off.

For now, the solution design does not incorporate optimization to eliminate either of these two artifacts.

Presumably, the next step would be vector analysis, and both of these artifacts can be more efficiently eliminated as part of vector analysis, which is currently beyond the scope of this project.

Implementation

The solution design, as described in this document has been implemented as a practical application in the form of an Android app, available for download to any valid Android platform at the following location:

https://play.google.com/store/apps/details?id=com.DragonAllen.MazeBot

This solution is a re-write of a win32 implementation designed in 2013. The “windows” implementation had little practical benefit, other than to illustrate the solution design.

As an Android app that runs on your phone, you can aMAZE your friends at parties!

Here is a link to a YouTube video that shows the application in action:

https://youtu.be/xmxnEz6fOeU

Real-World Applications

In addition to solving hand-drawn mazes, this solution could be leveraged in several different ways:

- GIS (Graphical Information System) data is effectively a node map overlaid on top of a graphical bitmap. For example, when you go to Google Maps and ask for directions, you are leveraging a very sophisticated GIS. MazeBot can potentially provide on-the-fly directions where GIS data is unavailable.
- On the electronic battlefield, GPS (Global Positioning System) data may be unavailable due to blocking or jamming. Mazebot can be trained to interpret bitmaps of land maps, and provide optimal routing vectors.
- MazeBot can be modified to provide optimal routing for automated machine tools.
- MazeBot is really fun at parties!

Future Works

MazeBot Enhancements

The following are possible enhancements to the solver algorithm

- **Resolve "join knob"**. The "join knob" is an artifact where the join cell has a higher metric than the surrounding cells, and a less-optimal cell, not part of the "path" solution has a lower metric, and is therefore considered part of the final solution.

  A final optimizer phase can look for cells with one neighbor, excluding start and end, and exclude them.

- **Resolve bifurcation.** A "path" cell with three neighbors represents a split in the path. This is caused by a rare situation where a disconnected "wall" artifact sits in the middle of a pathway and also happens to lie across the optimal path. The result is that both sub-optimal paths surrounding the artifact have an equal cost, therefore, both are valid solutions.

  To resolve bifurcation, the algorithm must identify equal path metrics, and then use a dead-end filling algorithm to eliminate the redundant pathway.

- **The optimization steps are a complete kludge**. In the original win32 implementation, I experimented with various 4 and 8 cell neighborhoods, various interim states, "parent cell" attribute, and the like, but all of those approaches yielded a sub-optimum outcome.

  Revisiting an approach based solely on cell attributes and neighbor count, even if it necessitates interim states, would be more flexible and less convoluted.

MazeBot's **key parameters** were experimentally-determined, and are implemented as constants within the application. A future version of MazeBot might implement a control panel so that the user can adjust these values:

- Cell Size
- Wall Threshold
- Contrast Neighborhood Size

The initial version of MazeBot is designed to minimally demonstrate the ability to capture and solve a maze, implementing the described solver algorithm.

MazeBot is not intended to be a full-blown image editor, but some minimal image editing functions would make it more usable:

- Ability to add “red” start point and “blue” finish point
- Ability to crop the image
- Ability to define arbitrary walls
- Ability to surround the image with a black box
- Ability to erase artifacts

All of these can be implemented as simple touch-and-drag functions.

Currently, the output layer only performs one function – it displays the raw maze bitmap with an overlay of the final pathway. The output layer can perform many other functions, such as:

- Ability to save a bitmap of the “solved” maze
- Ability to save the data set, either of the entire grid, or just the pathway, as a file
- Add a vector analysis layer that's capable of “vectorizing” the final pathway

Other Works

The following are possible extensions of the logical approach used by MazeBot

- Dynamic image analysis, such as edge detection or pattern recognition
- MazeBot is NOT an “AI” (Artificial Intelligence) application, but the concepts taken from MazeBot could be useful AI building blocks.
- Multiple, independent, but interconnected cell layers could be used for more sophisticated processing. Each cell layer's current state and attributes would be available to the layer above and below it, allowing each layer to perform aggregate analysis of the layers below it.

In general, cell-based image analysis could be useful in many practical ways:

- Meteorology - Analyze time-lapse weather radar images to provide hyper-accurate, localized, short-term weather predictions, such as predicting tornados.

- Astronomy – Analyze time-laps telescopic images for motion and other changes
- Cartography – Analyze map images to generate node (GIS) data

Conclusion

In this document, we've demonstrated a practical method to solve hand-drawn mazes using cellular automata and state machine logic.

- The application architecture is defined in layers, where each layer performs a specific function and passes the output to the next layer.
- Image acquisition, optimization, and analysis are handled by the first few layers. Once the grid array of cells is properly initialized, it's then passed along to the solver.
- The solver implements cellular automation driven by a machine / cell state framework that includes code blocks tied to specific states.
- Each “outer” state of the solver defines a particular goal or phase of the machine, while the “inner” state logic drives the cell states.
- For a given machine state, cell transitions in each frame (grid sweep) are counted, and are known as “flips”. Zero flips indicates stable cell logic, and is used to govern overall aspects of the machine state.
- Each cell, in addition to its state, has a distance metric derived from its surrounding neighbors' highest state, plus 1.
- Once a connecting path is identified from start to finish, the retrace process eliminates all sub-optimum cells based on the distance or cost metric, resulting in a set of multiple possible solutions of equal distance or cost.
- Two optimization steps eliminate redundant solutions, yielding a one-cell-wdith path connecting the maze start to its finish.

Cellular automata coupled with state machine logic provides a simple and flexible framework for image analysis, and other distributed problem-solving applications.

To view MazeBot in action, visit the YouTube Video:

https://youtu.be/xmxnEz6fOeU

To download MazeBot to your Android device from Google Play, visit Google Play:

https://play.google.com/store/apps/details?id=com.DragonAllen.MazeBot