Generating Mazes Using Cellular Automata

Justin A. Parr
q3ej7ejsjb@snkmail.com
http://justinparrtech.com
Version 1.0, 11/23/2018

Executive Summary

Although there are existing Cellular Automata that produce “maze-like” results, they do not produce a complete maze in that, there is a discreet pathway between any two nodes, and every node is accessible.

This paper describes a method for creating true, random mazes using Cellular Automation.

Table of Contents

Overview of Existing Solutions

Maze CA

There is a Maze-like CA:

http://www.conwaylife.com/w/index.php?title=Maze

However, there are no guarantees that the output product is a complete maze.

Maze-Generating Algorithms

Existing Maze-Generating Algorithms track cells and walls, where each cell has four walls to start. Each algorithm differs slightly, but they all generally employ recursion.

https://en.wikipedia.org/wiki/Maze_generation_algorithm

Limitations of Existing Solutions

Existing cellular automation produces a highly-randomized output, but not one that is highly-ordered, nor one that could be considered (nor guaranteed to be) a “complete maze”.

Other maze-generating algorithms use recursion and other graph traversal techniques, which are computationally-expensive.

Background

This solution makes extensive use of Cellular Automation (CA). For more information, please read:

https://en.wikipedia.org/wiki/Cellular_automaton

For more information about **solving** mazes using Cellular Automata, please read:

http://justinparrtech.com/JustinParr-Tech/wp-content/uploads/SolvingMazes.pdf

Project Goals

The solution must:

- Draw a traditional, complete maze:
  - Produces rectangular hallways of uniform width, which turn and / or join at 90 degree angles.
  - No “jagged” hallways nor cavities.
  - A path must exist from any point of any hallway within the maze, to any other point within any other hallway.
  - The maze must be a spanning tree, where every possible node is included in the maze, without any cycles or circular routes between any new nodes.
  - The maze must be solvable using a string of simple NSEW directional commands to traverse between junctions and around corners.
- Be computationally-efficient.
- Produce a graphical, visual representation of the result set.

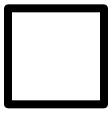
Solution Design

Approach

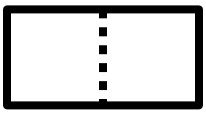
The solution uses a grid of cells, each of which can connect to exactly one of its neighbors, only if that neighbor is “connected” (part of the final set).

Maze-Generating Algorithm Concepts

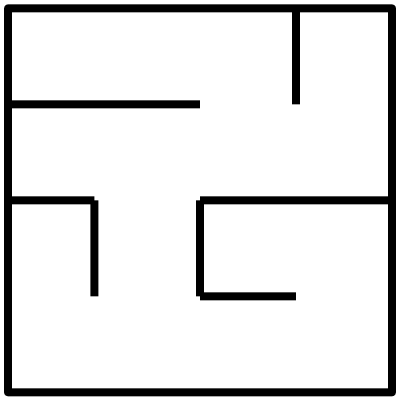
Conceptually, a “cell” is a spot in the maze where you could stand, and all of the cells begin in an isolated state, meaning that each has four walls.



As the algorithm progresses, cells are joined to neighboring cells by *removing* a common wall between two cells:

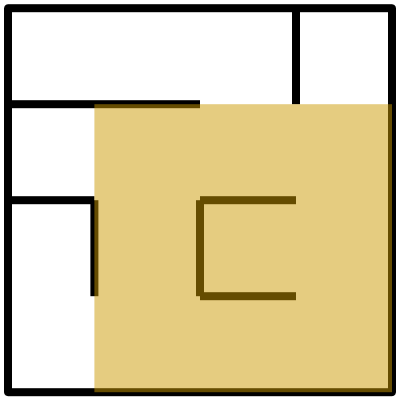


Eventually, all of the cells become connected:

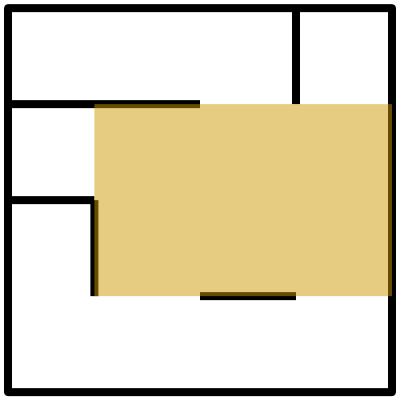


The outcomes we want to avoid are:

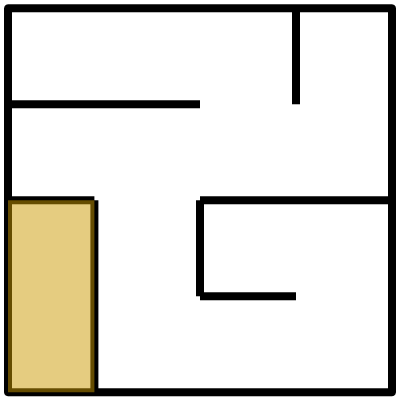
- **Cyclic path**, where a given cell can be reached by multiple routes (meaning: you could walk around in circles):



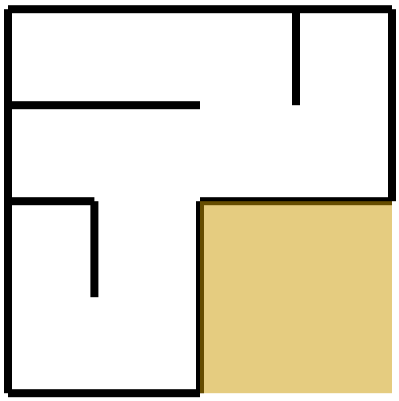
- **Cavity**, where multiple cells form an open space greater than 1 x 2.



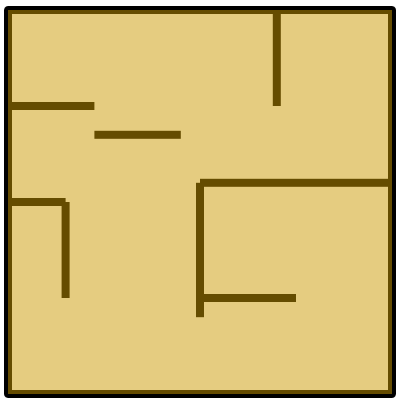
- **Isolated cells**, where one or more cells don’t connect to the main body.



- **Incomplete / Inefficient Space utilization**, where some cells are excluded.



- **Jagged / non-uniform hallways**, where walls and / or tunnels don't connect evenly.

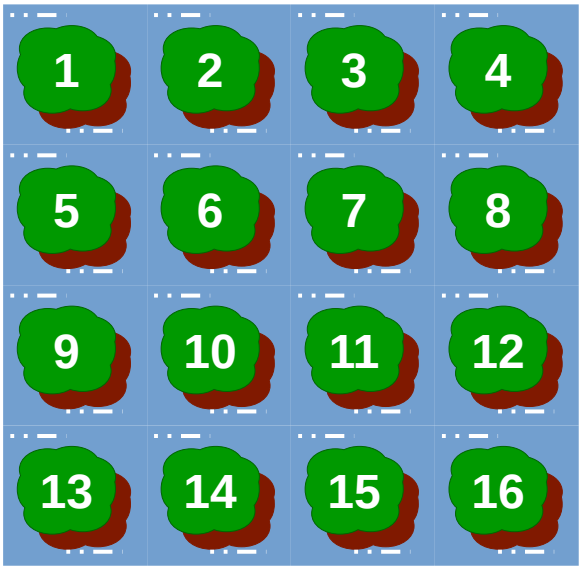


Conventional Approach

Most maze-generating algorithms start with a connected node, and search for nearby cells that can be joined. Once the maze generator reaches a dead-end, or some other limit is reached, it retraces back to a nearby node that has free cells nearby, and creates a branching path. This continues until every possible cell is converted in to a node.

This forward-looking approach requires recursion and complex data structures, used to track where it has been, where it needs to look next, and what it needs to do.

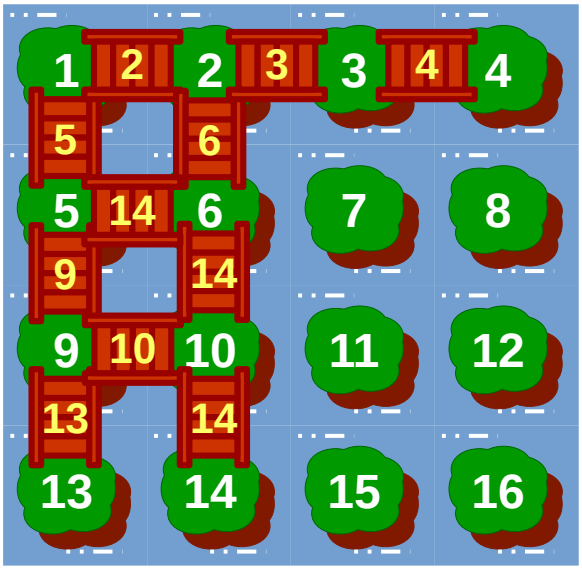
By metaphor, we can compare each cell to an island that is initially isolated.



Again, by metaphor, let's state that there is a castaway on each island, and every person needs to be rescued. Our seed location (1) is where the rescue boat will pick up our castaways, so we need to build bridges between islands so that every castaway can eventually traverse the maze to (1) where they will be rescued.

In a traditional maze-solving algorithm, one person is in charge of determining where to build the next bridge. He has to track the entire maze, where he has been, and who is left to rescue.

Conversely, if we let all of the castaways make their own decisions, we will definitely end up with loops, and maybe even isolated cells.

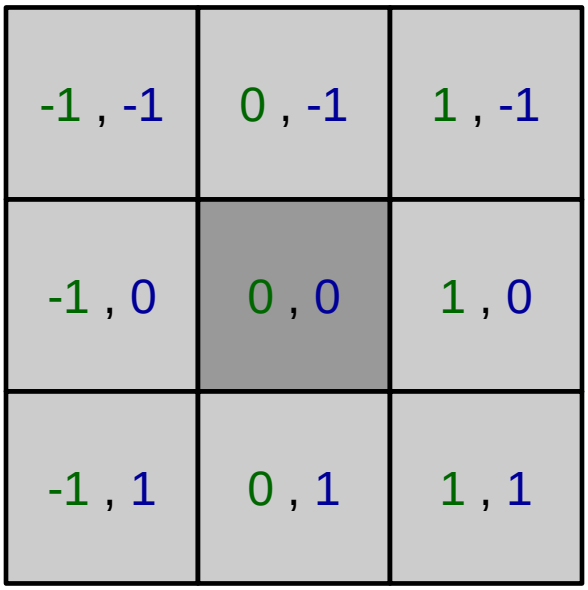


A Word on Vectors

There are two ways to scan all of your neighbor cells.

The first method is to scan your cell location, +/- some factor.

In this approach, we scan a range of cells, x+/-1 .. y+/-1 to form the 8-cell neighborhood:

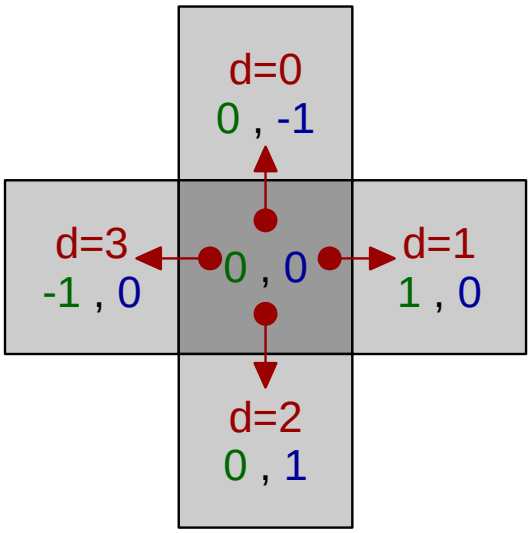


The problem with this approach is that there is no explicit relation to the originating cell. Also, we want to use a 4-cell neighborhood, which requires a bunch of exclusion logic.

The better approach is to use a vector table. Keep in mind that we are using a 4-cell neighborhood:

```
//build vector table
int xi[4], yi[4];
xi[0]= 0;  yi[0]=-1;  //NORTH
xi[1]= 1;  yi[1]= 0;  //EAST
xi[2]= 0;  yi[2]= 1;  //SOUTH
xi[3]=-1;  yi[3]= 0;  //WEST
```

In this approach, we have a list of vectors (0-3) and each vector has an x-offset and a y-offset.



By using a vector table, we have certain advantages:

- By stepping through the vector table, we scan ONLY the neighbor cells that we need.
- The originating cell is automatically excluded.
- We can create irregularly-shaped neighborhoods, by simply adding vectors for the extra neighboring cells. For example, we could add vectors for (-1,1) and (1,-1) to form a hex-shaped 6-cell neighborhood.
- Because each vector has an ordinality, it's easy to use modular math, to change or reverse direction.
- The vector can be stored for later use, if that particular neighbor is relevant.

Using a Bitfield to Store a Vector Map

If we elect NOT to use a vector table, each relevant neighbor must be tracked using its x and y offsets from the original cell.

Since both x and y offsets can have one of three possible values (-1, 0 or +1), each offset requires 2 bits, and therefore each neighbor requires 4 total bits (2 for the x offset, 2 for the y), requiring 4 * 4 = 16 bits to track all of it's 4-cell neighbors.

Conversely, with a use of a vector table, a cell can keep track of its neighbors by building a vector map. For example, in the CA described below, seed cells build a list of disconnected neighbors.

In the vector map, we can track each vector using one bit, which is either 1 if the corresponding vector is relevant, or 0 if not:

Vector	Bitfield
North (0)	2^0 = 1
East (1)	2^1 = 2
South (2)	2^2 = 4
West (3)	2^3 = 8

For example, a value of 11 (base 10) means that the cell has relevant neighbors to the North, East and West, but NOT to the South:

11 = 8 + 2 + 1, or 1011 in binary.

8 = 2^3 = **West**
2 = 2^1 = **East**
1 = 2^0 = **North**

We start with bitfield=0, and as we scan each neighboring cell, if we find that the neighbor is relevant, we add it to the bitfield using OR:

Assuming “v” is our current vector (0 to 3):

bitfield = bitfield || (2^v)

Note: || = Bitwise OR

The resulting bitfield value can be stored as part of the cell’s state, and retrieved later.

Vectors and Directions

A vector implies a direction, and without vectors, it’s difficult to conceptualize relative changes in direction, such as:

- Turn Left
- Turn Right
- Reverse

For example, if we have only x and y offsets, we could reverse them to find the originating cell (x*-1 , y*-1), but we either have to have complex math or complex logic to figure out how to turn left or right.

However, with vectors, these operations become trivial, using modulus 4 arithmetic.

For a given vector “v”:

- Turn left: v’ = (v + 3) mod 4
  Note: in modular math, (v + m - 1) is the same as (v - 1). Since m, the modulus, is 4, (v+3) is the same as (v-1).
- Turn right: v’ = (v + 1) mod 4
- Reverse: v’ = (v + 2) mod 4

Let’s see how this works:

Vector	Turn Left	Turn Right	Reverse
0 (North)	$(0+3)_4 = 3_4 =$ **3 (west)**	$(0+1)_4 = 1_4 =$ **1 (east)**	$(0+2)_2 = 2_2 =$ **2 (south)**
1 (East)	$(1+3)_4 = 4_4 =$ **0 (north)**	$(1+1)_4 = 2_4 =$ **2 (south)**	$(1+2)_4 = 3_4 =$ **3 (west)**
2 (South)	$(2+3)_4 = 5_4 =$ **1 (east)**	$(2+1)_4 = 3_4 =$ **3 (west)**	$(2+2)_4 = 4_4 =$ **0 (north)**
3 (West)	$(3+3)_4 = 6_4 =$ **2 (south)**	$(3+1)_4 = 4_4 =$ **0 (north)**	$(3+2)_4 = 5_4 =$ **1 (east)**

The CA Approach

Because a CA looks at the entire maze, we can define a few simple rules that allows each cell to make its own decision about how to attach itself to the maze.

- Every cell starts in the “disconnected” state.
- We pick a seed, at random. Ultimately, every cell will connect directly or indirectly to the seed. We can’t define multiple seeds, as this would result in multiple, disconnected pathways within the same maze.
- The seed picks one disconnected neighbor cell at random and connects to it. The new cell becomes the seed, while the old cell, within some probability, either remains a seed (branching logic), or becomes connected (final state).
- If a seed cell has no disconnected neighbors, it becomes connected (seed dies).
- If all seeds die, and there are still disconnected cells, then, within the branching probability, any connected cell neighboring a disconnected cell becomes a seed.

- This repeats until all cells are connected.

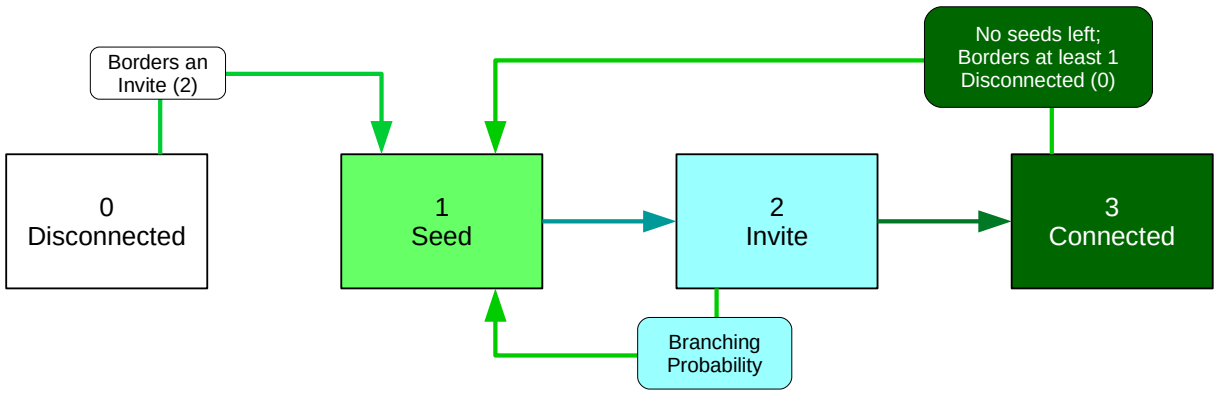
Cell Structure

Each cell has the following attributes:

Attribute	Description
State	Cell state (0 to 4)
ConnectVector	Vector (0-3) pointing to parent cell
InviteVector	Vector (0-3) pointing to the invited cell.
Neighbors	Bitfield (0..3) indicating a seed cell's disconnected (eligible) neighbors

State Machine Definition

Here is a state machine representation:



Cell State	Actions
0 (Disconnected)	Initially, all cells start in the disconnected state. Disconnected cells look for an invitation from a neighbor in state 2 (Invite). If a state 2 neighbor cell's InvitationVector is opposite of the vector pointing to the neighbor cell, then the neighbor is pointing to this cell, and this cell performs the following tasks: 1. The neighboring cell is considered the parent of this cell. 2. Store a vector pointing to the parent in cell.ConnectVector. 3. Change to state 1 (Seed).
1 (Seed)	Initially, one cell is picked at random to be a seed. Seed cells perform the following tasks: 1. Look at each neighbor, and build a vector map listing neighbors in the 0 (disconnected) state. Each of these is a candidate. (Store bitfield in cell.Neighbors) 2. Pick one of the candidates at random. 3. Store the vector pointing to that candidate in cell.InviteVector 4. Change to state 2 (Invite).
2 (Invite)	This is a flag which tells the invited neighbor in state 0 (disconnected) to change states and become the new seed (1). This cell assumes that the neighbor received the invitation, and ignores the neighbor's state. Based on a predetermined branching factor, it picks a random number, and either changes state to Connected (3), or goes back to being a Seed (1).
3 (Connected)	There are two possible sub-states: If there is at least one, live seed, this cell does nothing. If there are NO live seeds, AND if this cell borders at least one disconnected cell, then this cell will conditionally become a seed (1), based on the branching factor.

	For all practical purposes, Connected (3) is the final state, unless all seeds die.

Constants

The following constants dictate constraints about how the seeds propagate to disconnected cells:

Constant	Effect on Maze

BranchProbability	*Range 0-100*
	
	Specifies the probability (percent chance) that a state 2 (Invite) will return to a seed (1) state, effectively creating a branch.
	
	We define a function r() that returns a random integer between and including 0 to 100.
	
	`if r()>BranchProbability,`
	`  cell.State = 3  //connected`
	`else`
	`  cell.State = 1  //revert to seed`
	
	If all seeds die, this is also the probability that a “connected” (state 3) cell, who neighbors a disconnected cell, will become a seed.
	
	If the value is too low, there are too few branching pathways. The resulting maze has a few, long hallways, with many tiny dead-ends.
	
	If the value is too high, the resulting maze looks like a tree-like structure, as there are too many branching pathways.
	
	Experimentally, the optimum value has been determined to be 5. This represents a 5% chance that a seed will create a branching pathway.

TurnProbability	*Range 0-100* Specifies the probability that a seed will change direction.

Recall that, if a seed borders multiple disconnected neighbors, it will pick **one** of them to invite to become the new seed.

Let's define a function, nd() that returns a random integer between and including 0 to 3, representing a random direction.

```
d=nd()
while (cell.Neighbors && (2^d)==0)
      d=nd()
```

This will continue to loop until d holds a direction that appears in the cell's neighbor list.

Without directional persistence, the resulting maze has lots of twists and turns, but the chance that a seed could reach a dead-end increases significantly.

We can implement directional persistence by copying the cell's parent vector, and reversing it. We can also introduce a random turn probability. In the following, nd() is defined as above, and r() is a function that returns a value from 0 to 100 inclusive.

```
if r()>TurnProbability,
      //Random, 0-3 - Turn.  Maybe.
      d=nd()
else
      //Reverse of parent vector - go straight
      d = ( cell.ParentVector + 2 ) % 2

// In either case, if we can't move in the
// chosen direction, keep picking until we
// pick a valid direction.
while (cell.Neighbors && (2^d)==0)
      d=nd()
```

Experimentally, the optimum value has been determined to be 10, representing a 10% chance that a seed will randomly change direction.

Execution

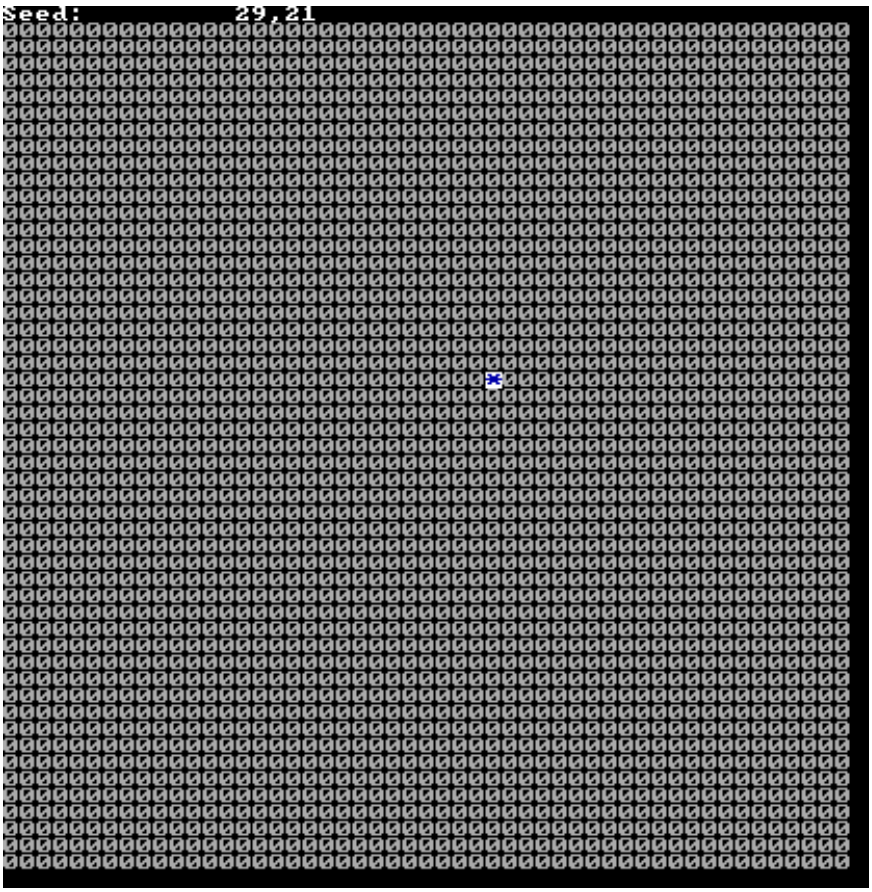
The following screen shots depict a sample execution within a 51 x 51 array.

Colors indicate the state of each cell, while the number displayed is the vector to its parent cell.

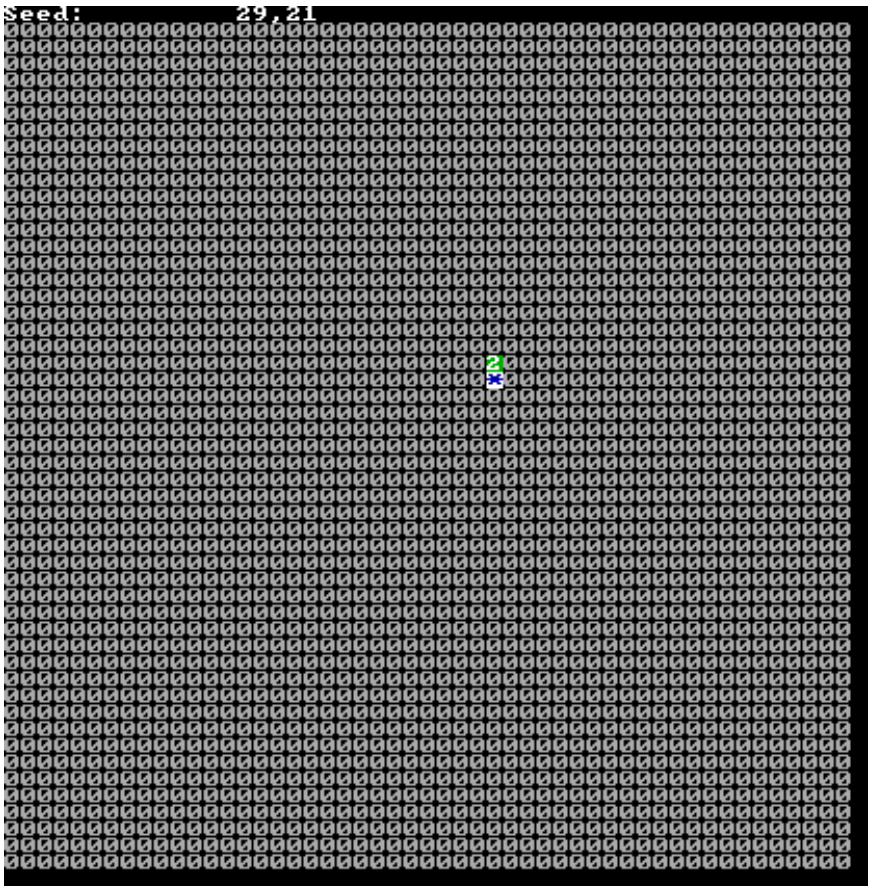
Cell State / Color
Disconnected (0)
Seed (1)
Invite (2)
Connected (3)

Direction
0 North
1 East
2 South
3 West

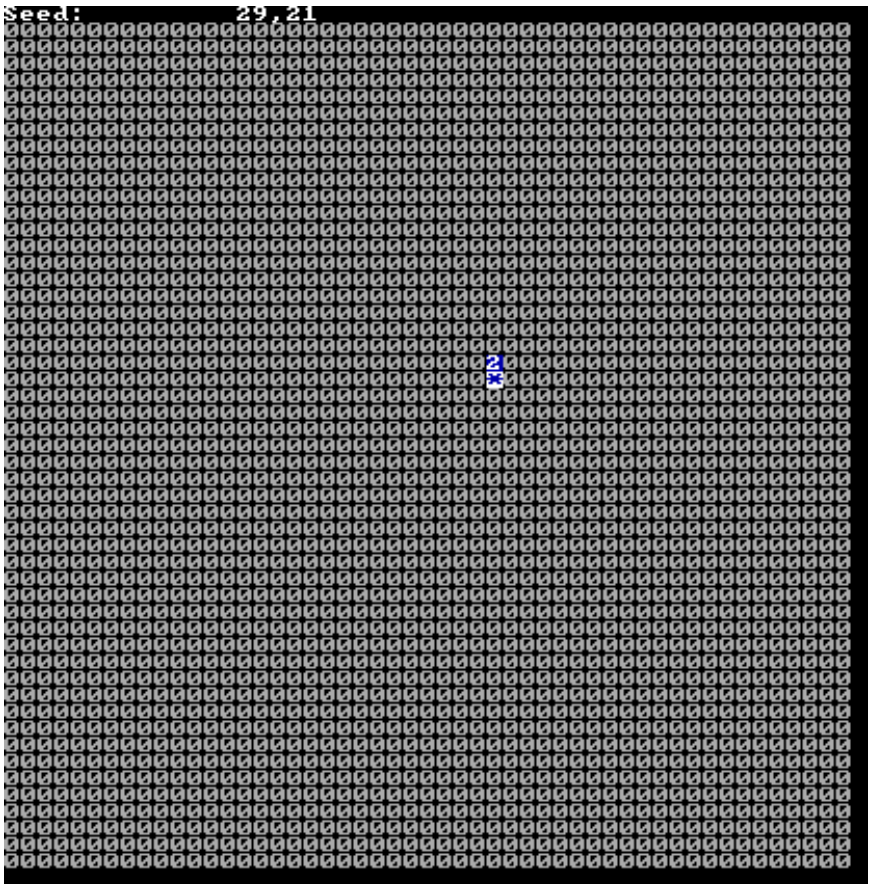
In the beginning, every cell is in the ‘disconnected’ state, except the seed.



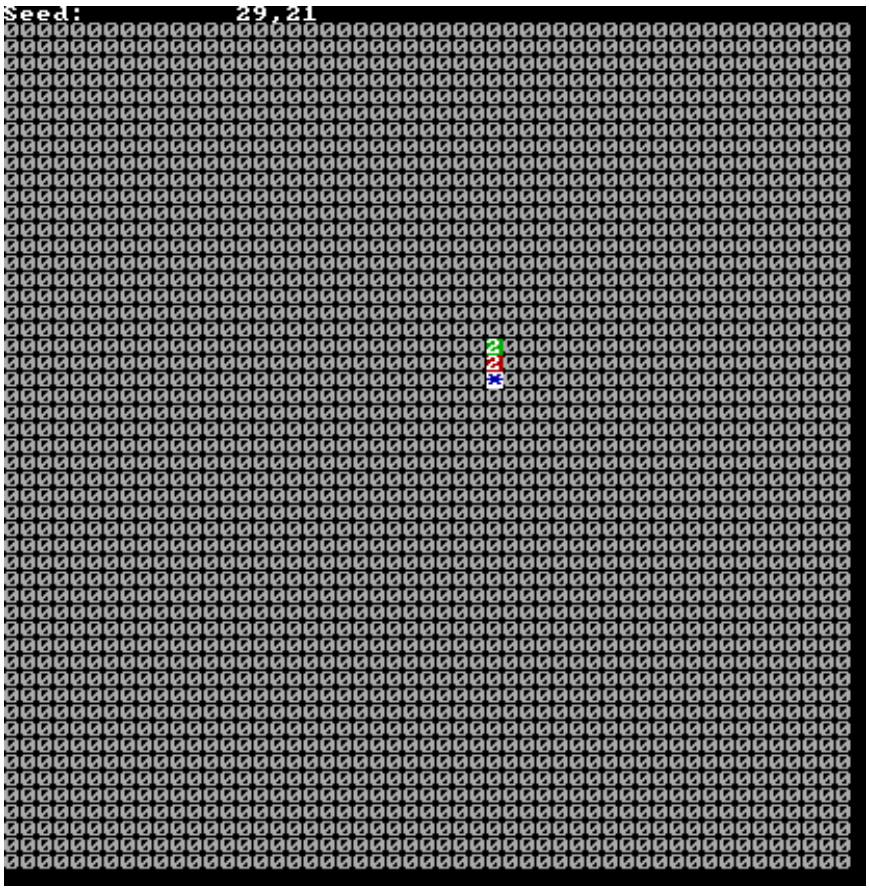
We see all zeros because the parent vector for the disconnected cells hasn’t been set yet.



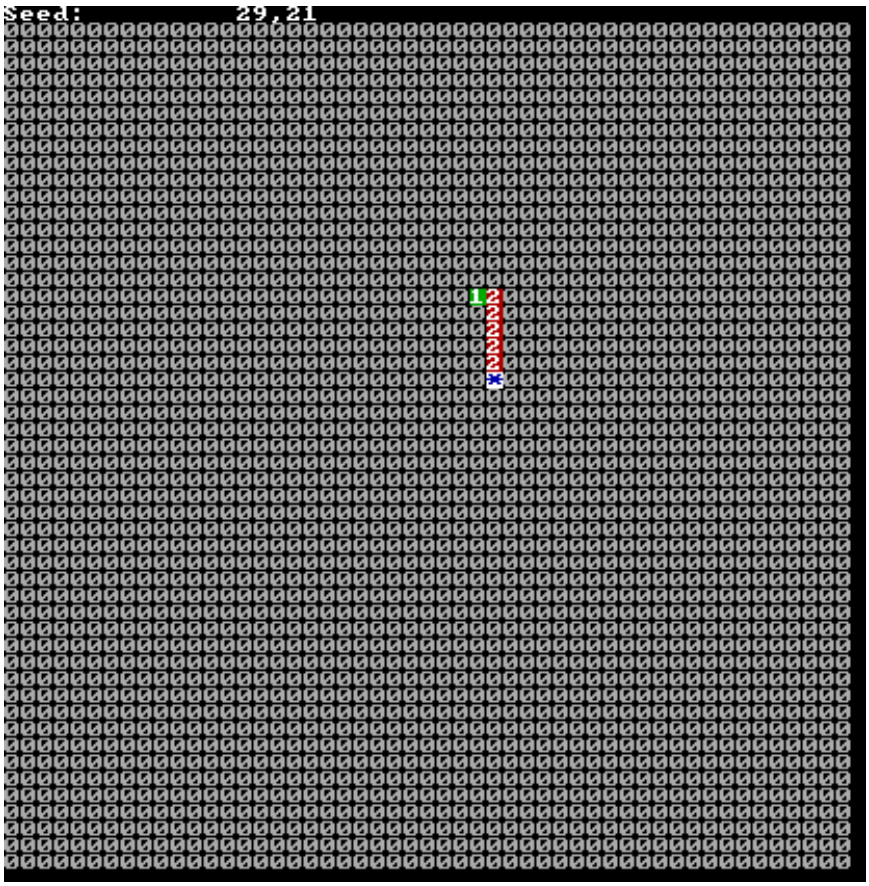
As the seed cell reaches the ‘Invite’ state, we see its neighbor cell accept, and switch to the seed state. Note that the parent vector, “2” points South, to the seed.



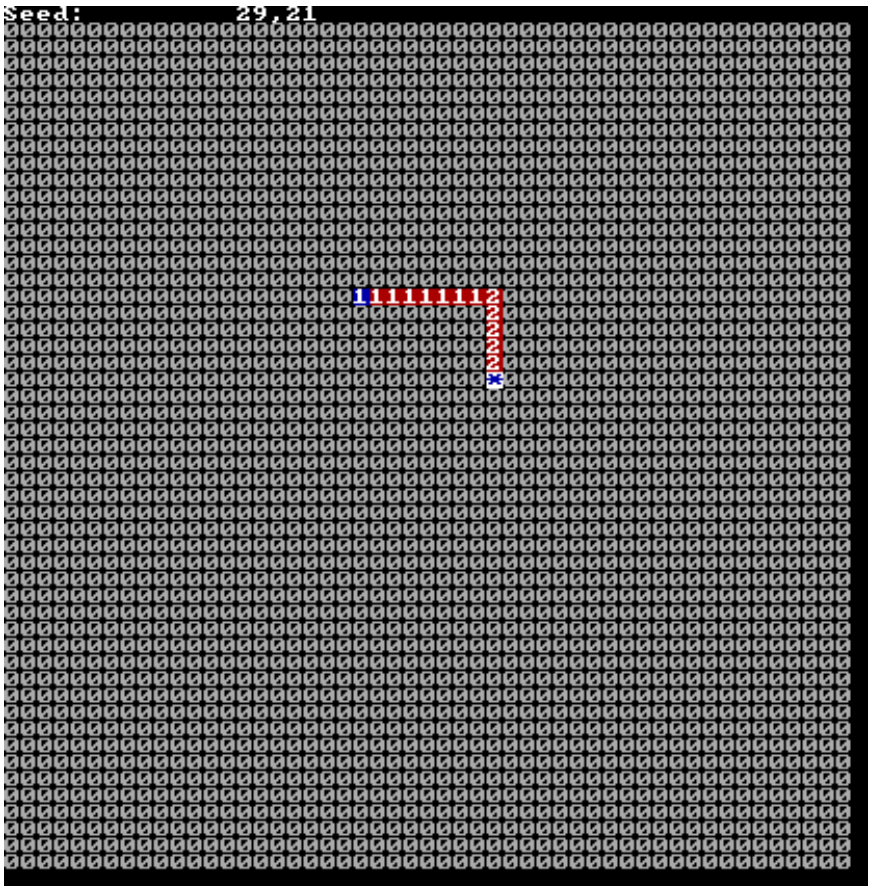
The seed switches to the ‘Invitation’ state, which will trigger its neighbor to connect, in the following frame.



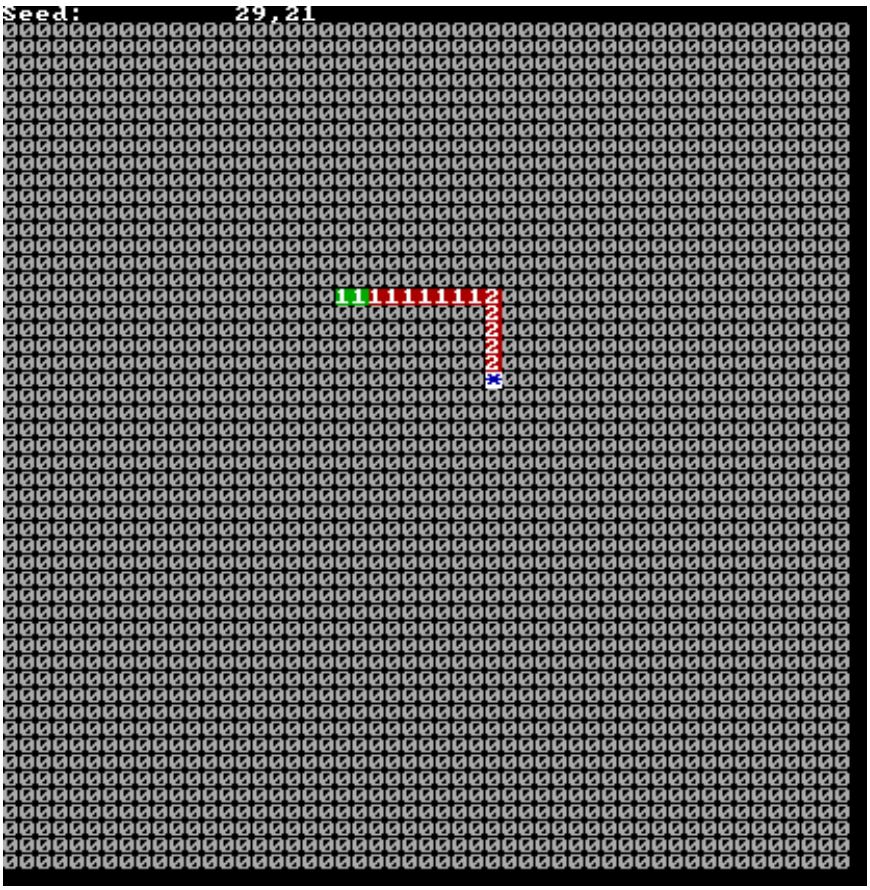
The new seed is connected via its parent vector to the old seed (south). The old seed switched to ‘Connected’ state, where it will remain unless all seeds die out.



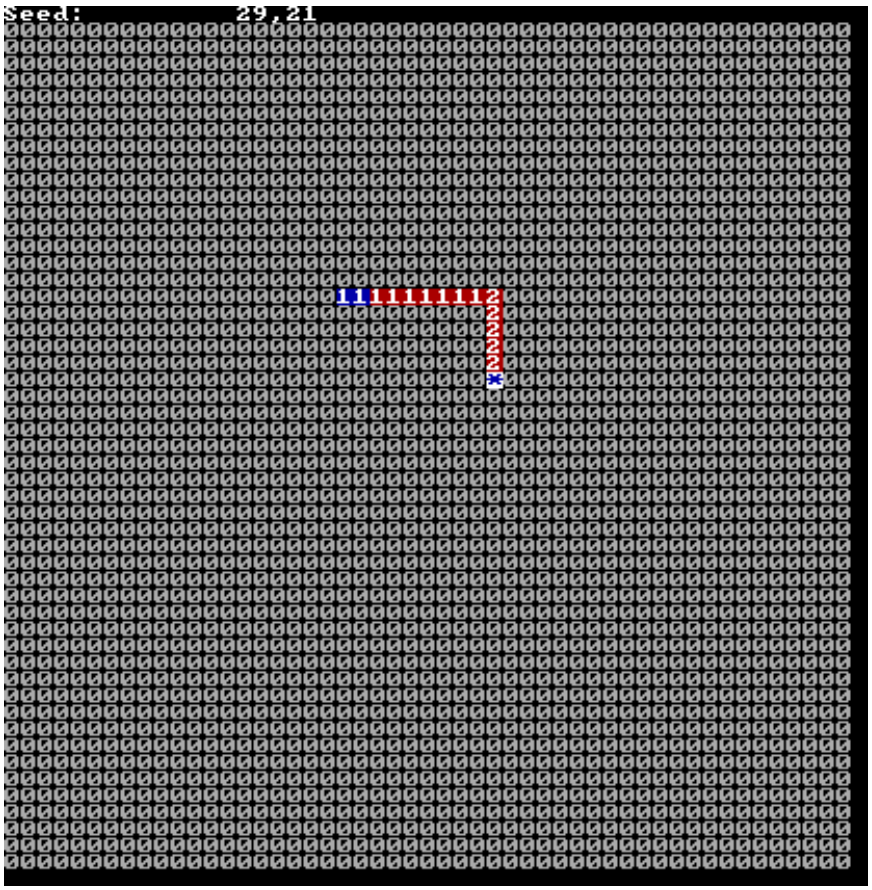
Directional persistence keeps the seed heading in the direction of its predecessor, until the turn probability is met.



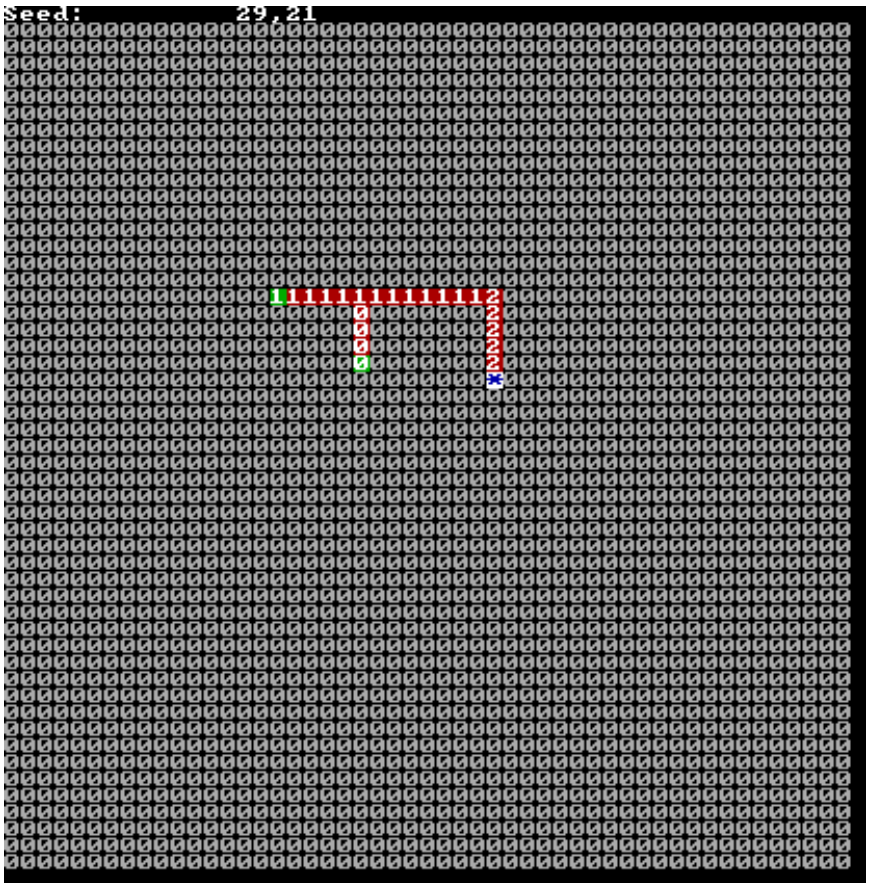
At this point, the seed is about to branch. We see the cell reach the ‘invite’ state.



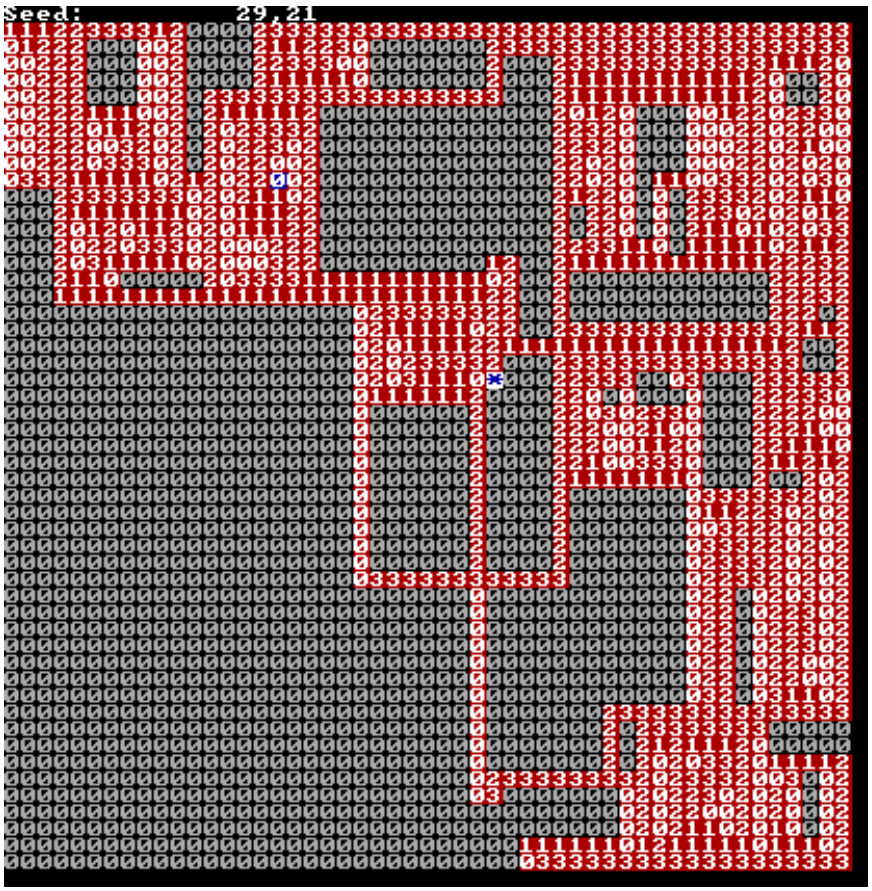
In the very next frame, we see the neighbor switch to the ‘seed’ state, but the original cell also switches back to the ‘seed’ state, rather than ‘connected’. At this point, there are two active seeds, creating a branch.



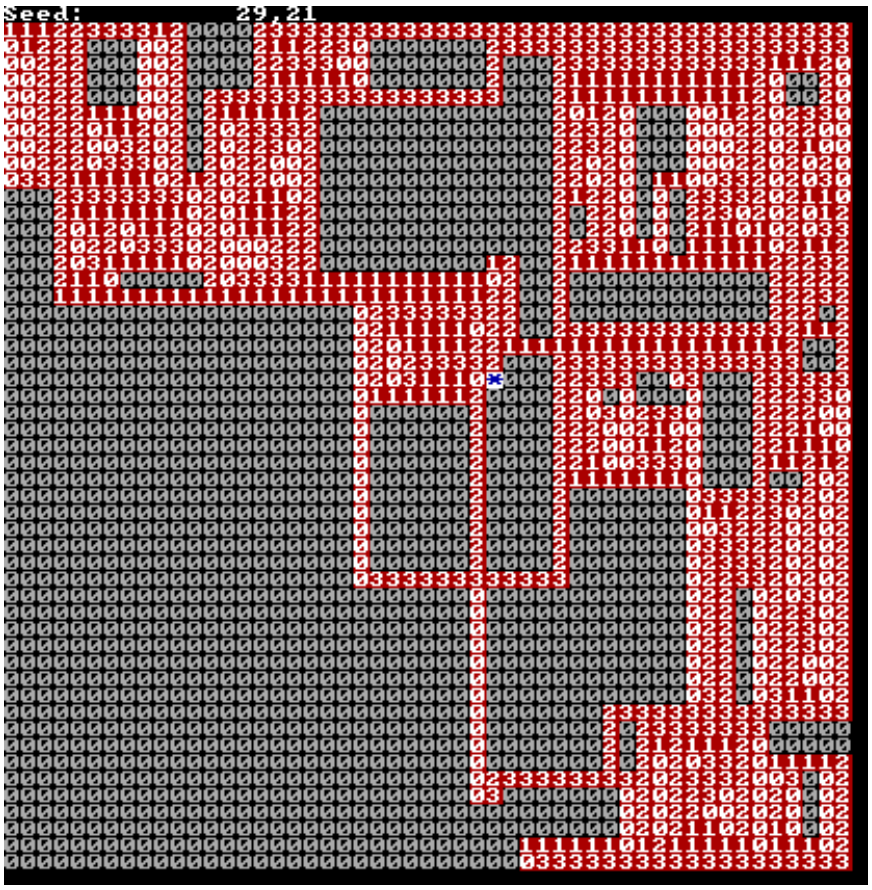
Both seeds invite one of their respective neighbors, and the process continues.



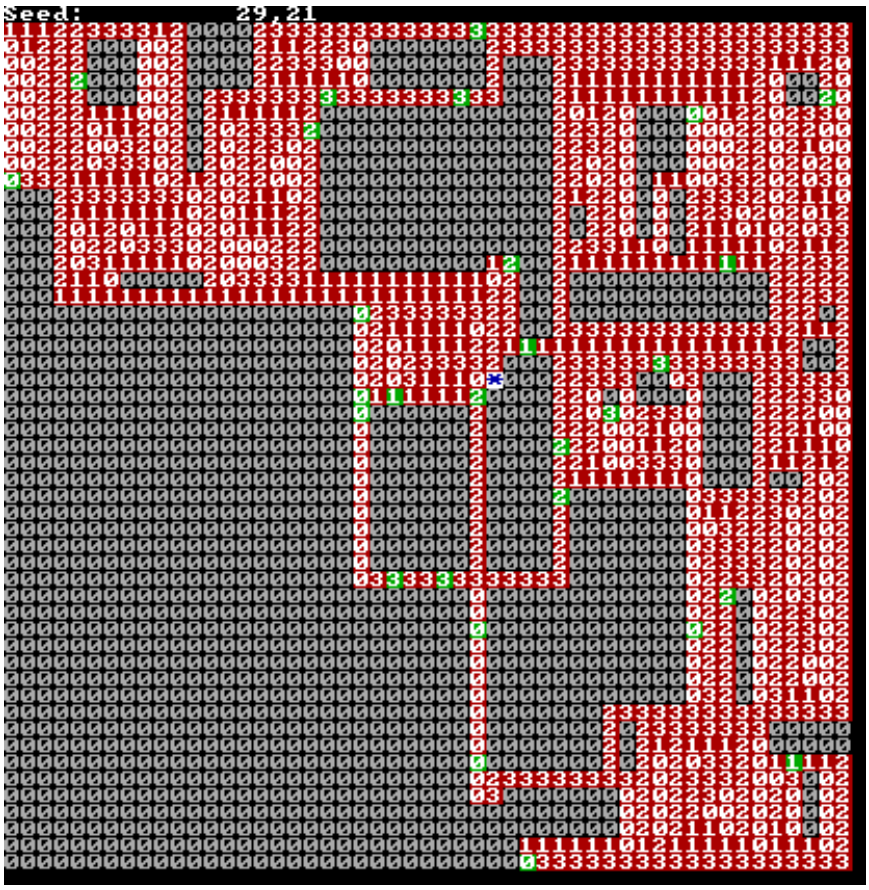
Several frames later, each seed is propagating independently.



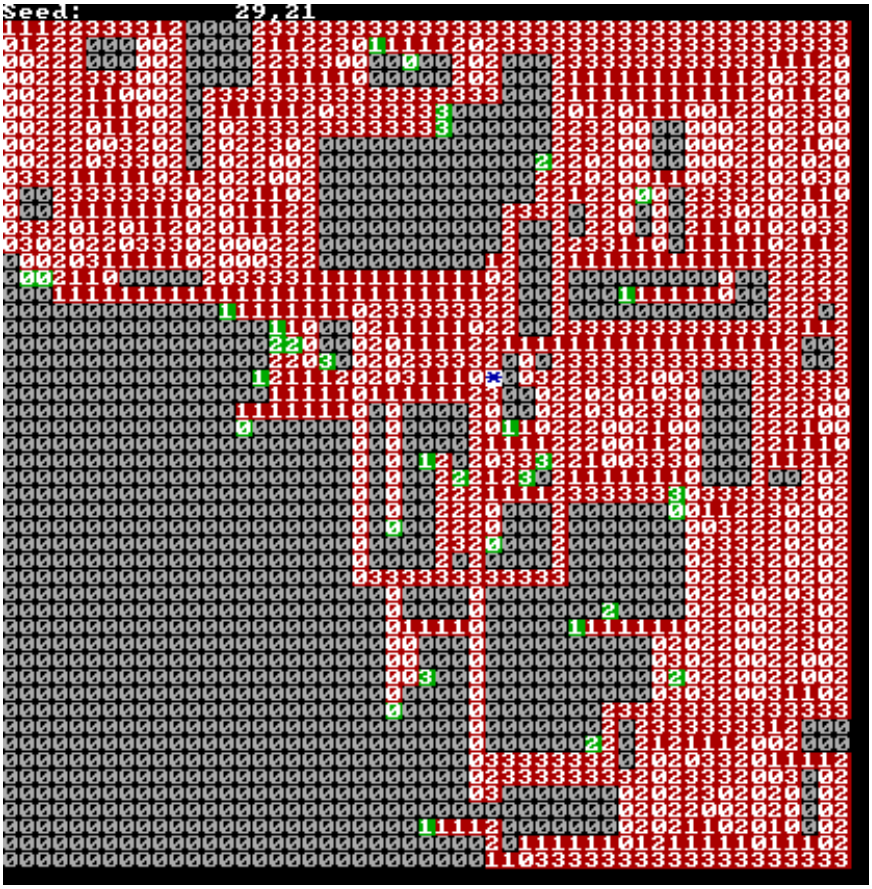
After about 300 more frames, we see the last seed about to die out (upper-left), as it tries to issue an invitation, but has no neighbors to invite.



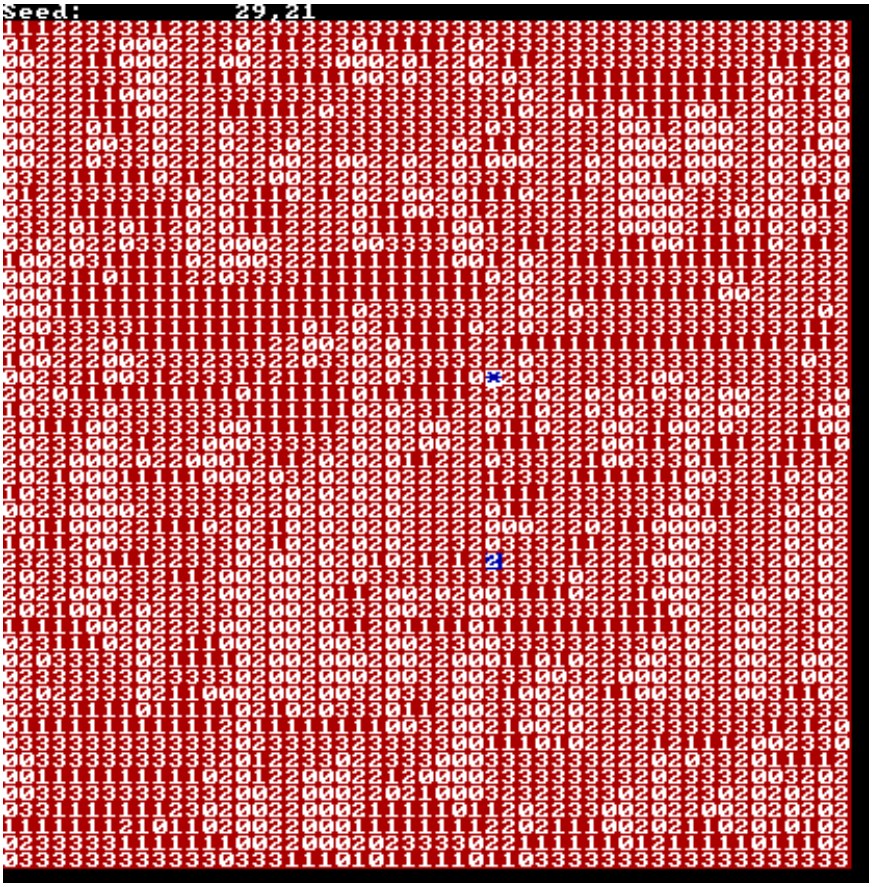
In the very next frame, all seeds are dead, as the last seed switches to ‘connected’.



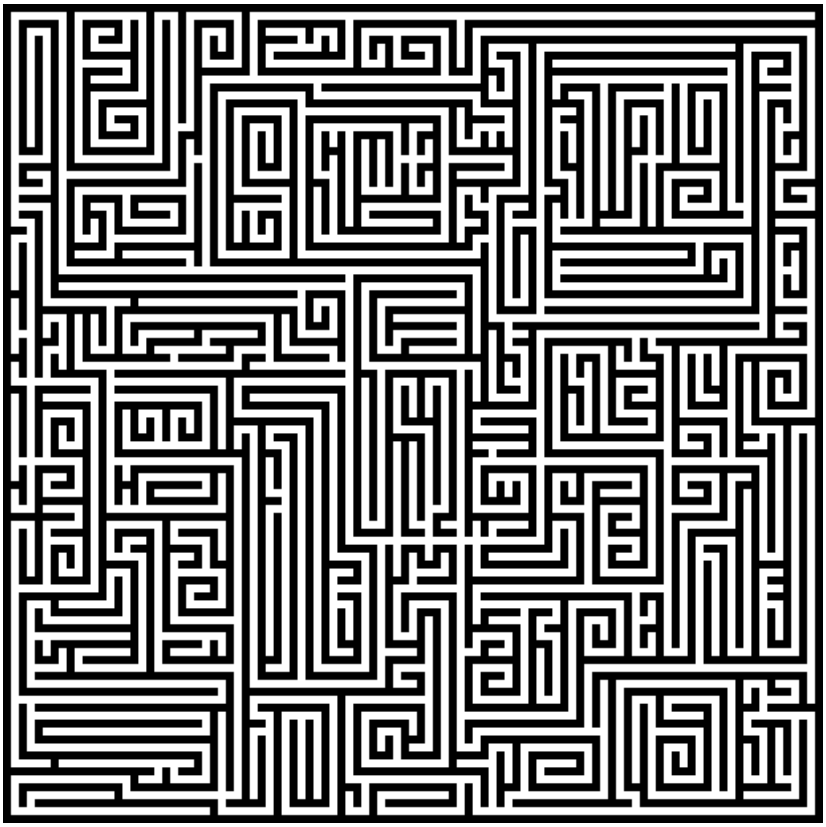
If all seeds die, but there are still disconnected cells left, this triggers a fail-safe, where some ‘connected’ cells which are neighbors to a disconnected cell, change back to ‘seed’ state.



The new seeds continue to propagate, and the cycle continues.



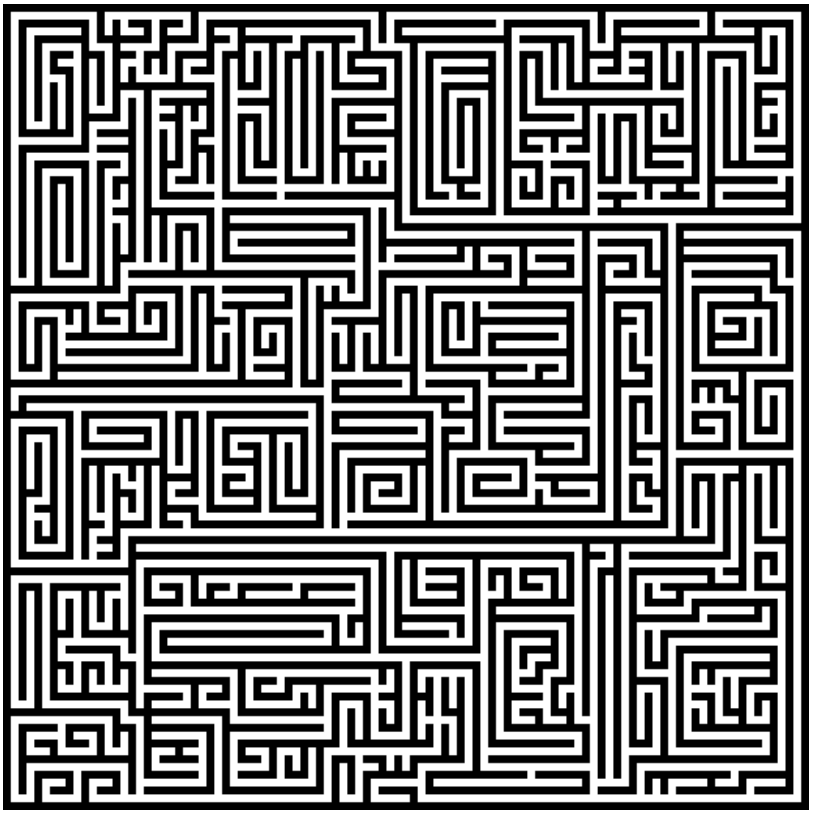
As the last seed dies, and there are no more disconnected cells, the maze is complete. If you follow the chain of parent vectors from any cell, they will eventually lead to the original seed (blue *).



Here is a bitmap of the resulting maze.

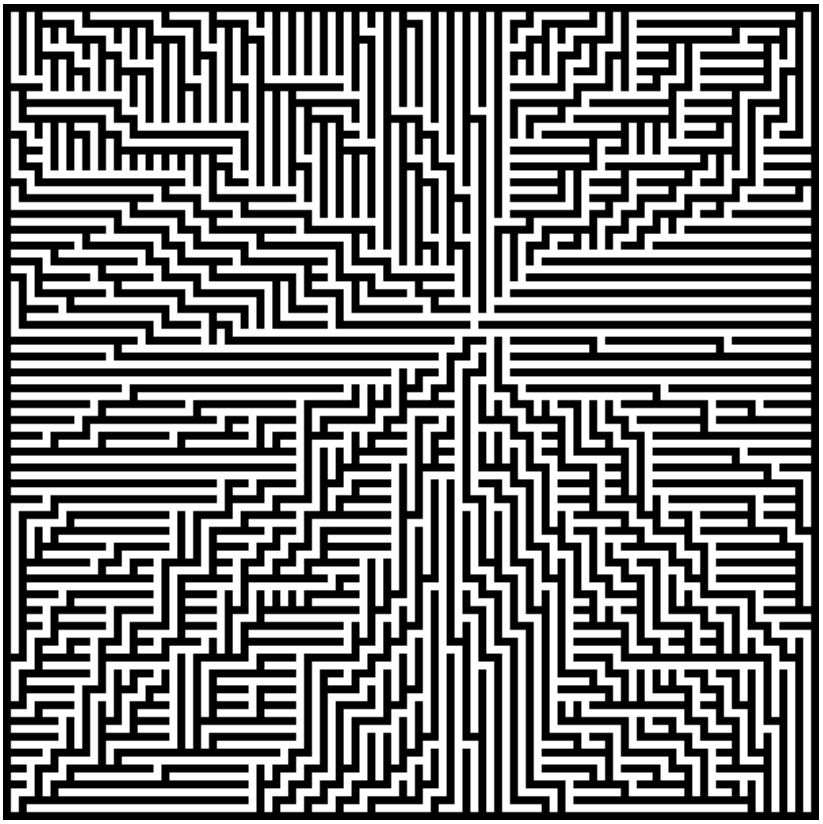
Effects of Optimization

Normal Result



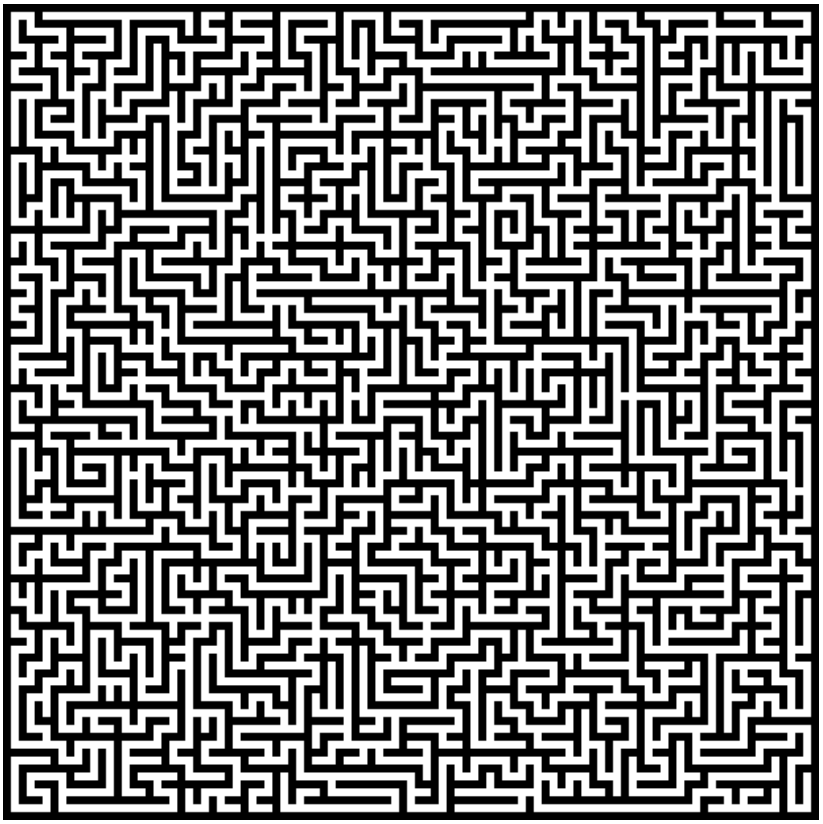
- Long hallways
- Branching pathways vary in length and direction
- Lots of parallel pathways
- Swirls and spirals

Too Many Branches



- Tree-like branching pattern
- Seed location is obvious
- Individual pathways don’t turn or deviate

Too Many Turns

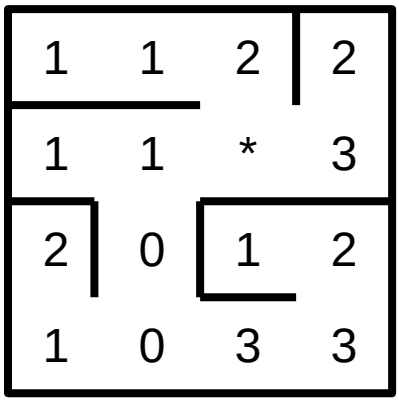


- Lots of twists and turns
- Seeds often hit a dead-end, resulting in shorter branching pathways
- “Ramen Noodle” effect – it’s more confusing than challenging.

Rendering

One of the more interesting aspects of this solution is that it produces a result set that consists of a set of cells, each linked to its parent, in a chain, which eventually ends at the original seed cell.

Every cell except the original seed has a parent, and therefore a parent vector.



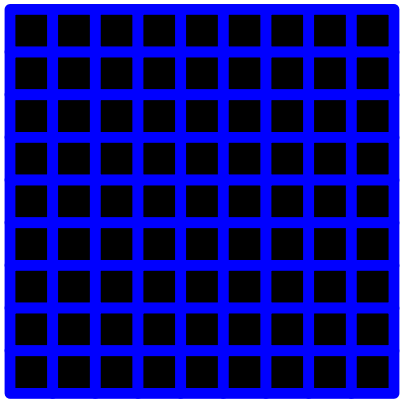
We can pick any cell at random, and walk the path back to the seed. For example, if we pick the southwest corner, its vector is 1 (East). We move East, then North, then North, then East, to the seed.

Rendering is the process of converting the array in to an actual picture of a maze, and there are two obvious methods to do this.

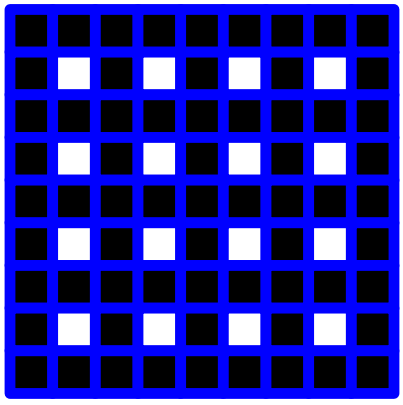
Method 1: Connect the Cells

We can view each cell as part of a grid, where the hallway and cell walls are each of equal width.

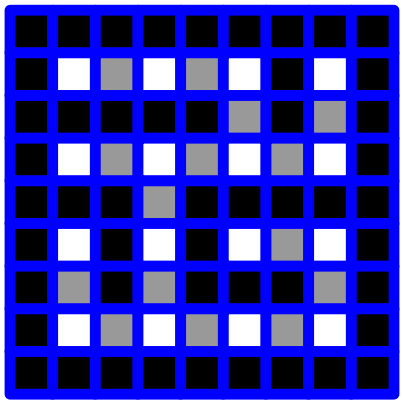
1. Start with all walls (solid black bitmap)



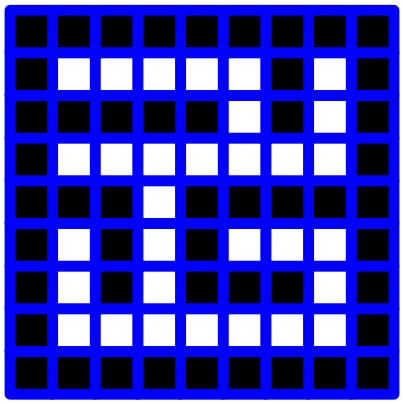
2. Render every cell as a white square, separated on all sides by black squares.



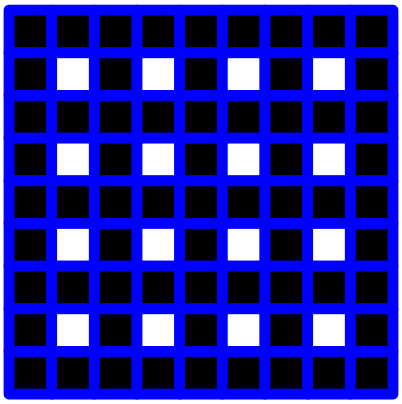
3. Use the parent vector of each cell to erase the adjoining walls.



The final rendered result:



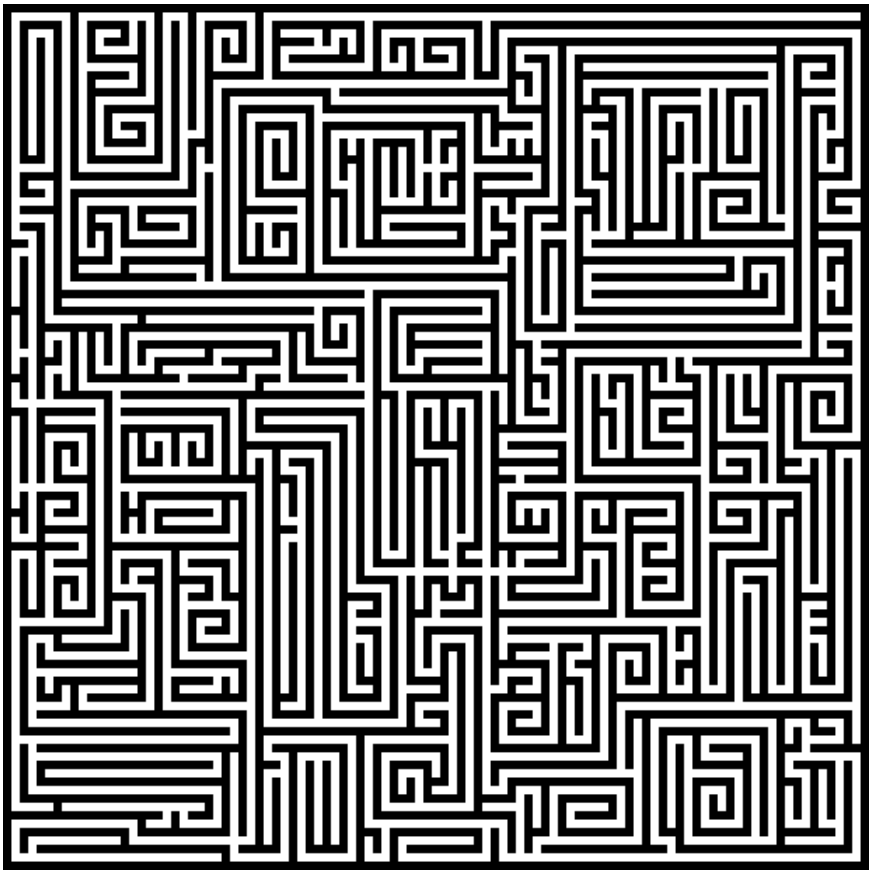
Calculating the size of the bitmap as well as the position of each cell is based on defining a block size, which essentially becomes the width in pixels of both walls and hallways.



```
//define block size in pixels
blockSize=5

//each “cell” block has a “wall” block to its
// left, and then we need one more column on
// the right, to form the last cell’s right-hand wall:
bmpWidth=(mazeWidth * 2 + 1)  * blockSize
```

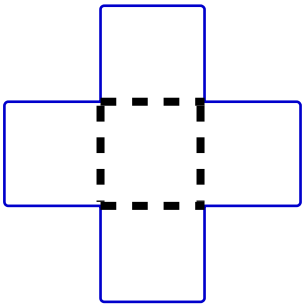
```
//bitmap height is calculated the same way
bmpHeight=(mazeHeight * 2 +1 ) * blockSize

//For each cell (x,y), we calculate (x1,y1) and
//(x2,y2) within the bitmap

x1=(x*2+1) * blockSize
x2=x1+blockSize-1
//ensures that each cell is exact

y1=(y*2+1) * blockSize
y2=y1+blockSize-1

//We would then draw a solid white rectangle
//on the bitmap from (x1,y1) to (x2,y2)

//To calculate the position of the adjoining cell,
//we use our vector table, xi[] and yi[]

x1=x1 + blockSize * xi[cell.ParentVector]
y1=y1 + blockSize * yi[cell.ParentVector]

//Now, recalculate (x2,y2)
x2=x1 + blockSize-1
y2=y1 + blockSize-1
```

Here is an example of a maze rendered this way:



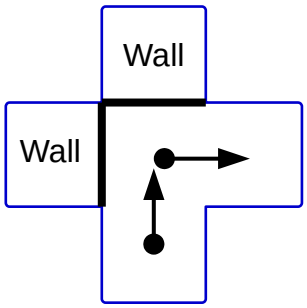
Method 2: Draw the Walls

We start by assuming that each cell borders all of its neighbors, and ONLY draw a wall if one is needed:



For each neighbor cell, we draw the adjoining wall if:

- cell.ParentVector does NOT point to it, AND
- the neighbor’s parentVector does NOT point to us



An easy way to track all of this, is to make a complete pass through the maze, and store wall information in cell.Neighbors.

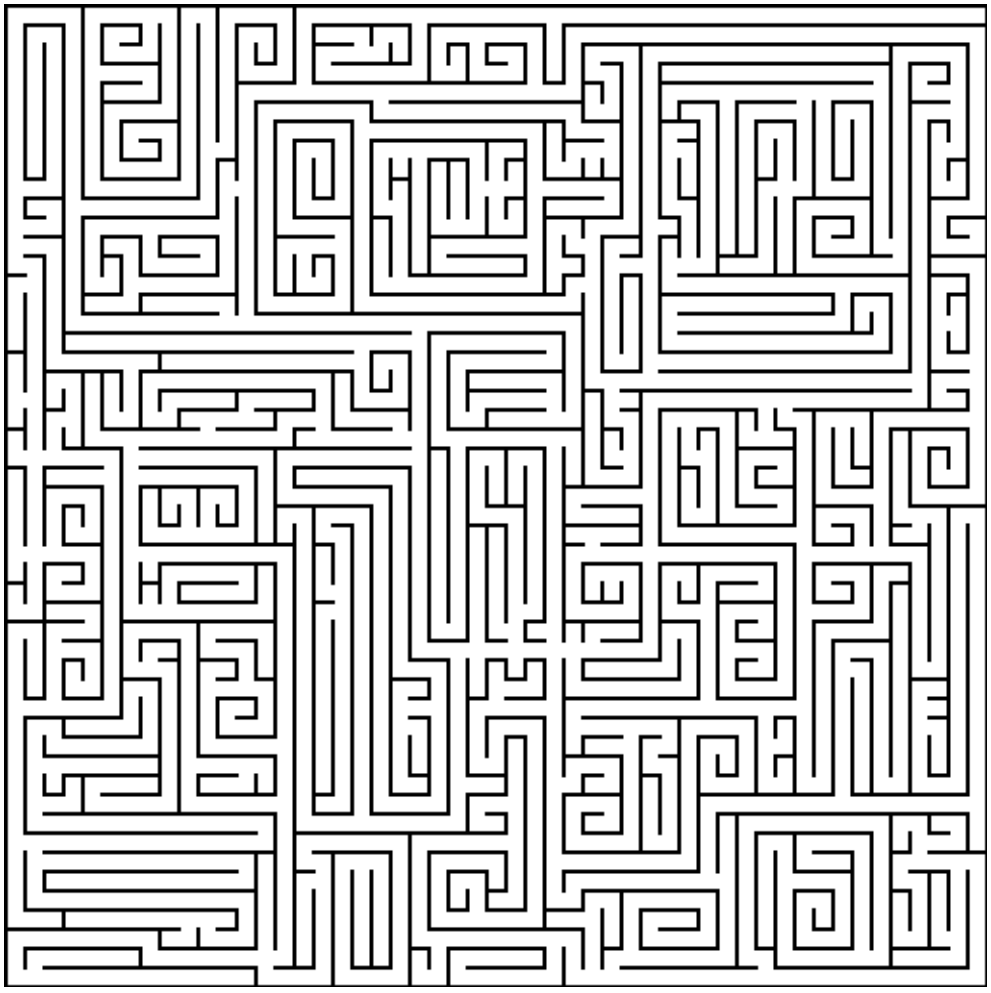
```
//We can use cell.Neighbors to track walls,
//so we start with all walls defined:
cell[x,y].Neighbors=15

//Remove walls adjoining parent and child cells

//Remove parent:
cell[x,y].Neighbors = cell[x,y].Neighbors -
2^cell[x,y].ParentVector

//Let d be the neighbor direction, so we
//loop d from 0 to 3

nx=x + xi[d]
ny=y + yi[d]

//Remove child:
//Child cell has the opposite perspective
nd=(d+2) % 4

//If neighbor cell's parent vector points
//to me, remove adjoining wall
if (cell[nx,ny].ParentVector == nd)
     cell[x,y].Neighbors = cell[x,y].Neighbors - 2^d
```

Rendering the maze is based on a fixed cell size. We calculate the bitmap dimensions, as well as each cell's position, and each wall's position based on the cell size.

```
//cellSize in pixels is used to calculate
//bitmap size and cell position
cellSize=10
```

```
bmpWidth=cellSize  * mazeWidth  + 1
bmpHeight=cellSize * mazeHeight + 1

//As we loop through the maze, (x,y) is the
//current cell
x1=x * cellSize
y1=y * cellSize
x2=x1 + cellSize
y2=y1 + cellSize

//Look at our Neighbors flag
//Assume drawLine(x1,y1,x2,y2) draws a black
//line between the two points
//North wall
if cell[x,y].Neighbors && 1 drawLine(x1,y1,x2,y1)
//East wall
if cell[x,y].Neighbors && 2 drawLine(x2,y1,x2,y2)
//South wall
if cell[x,y].Neighbors && 4 drawLine(x1,y2,x2,y2)
//West wall
if cell[x,y].Neighbors && 8 drawLine(x1,y1,x1,y2)
```

Here is the same maze from above, rendered using this method:

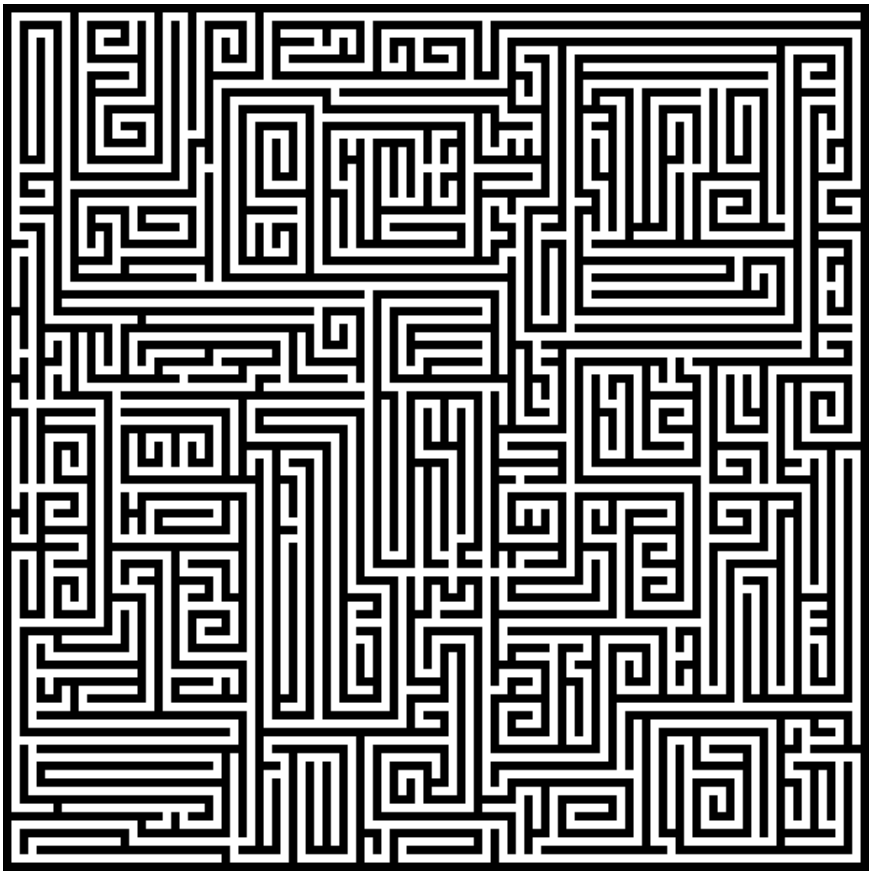


Solution Analysis

Referring back to the design criteria, let’s analyze the solution.

Draw a Traditional, Complete Maze

Here is an example of a maze, generated using this algorithm:



The maze above has these characteristics:

- Hallways of uniform width, which turn and / or join at 90 degree angles.
- No “jagged” hallways nor cavities.
- A path exists from any point of any hallway within the maze, to any other point within any other hallway.
- The maze is a spanning tree, where every possible node is included in the maze, without any cycles or circular routes between any subset of nodes.
- The maze is solvable using a string of simple NSEW directional commands to traverse between junctions and around corners.

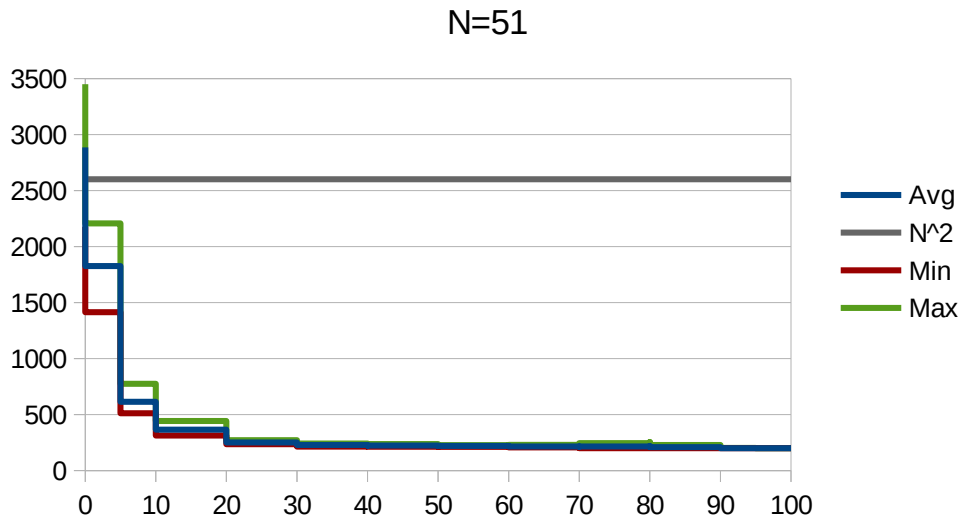
By these criteria, as we can see, the algorithm produces a complete maze.

Be Computationally-Efficient

The transfer of the seed state from a seed cell to its neighbor requires two cycles.

- If the branching factor were 0 (0% chance of creating a branch), and assuming that there are no collisions, the algorithm would fill the entire maze in (c - 1) * 2 cycles, excluding the seed, where c is the number of cells (width * height). If we assume n = width = height (square grid), the cycles required would be **2 * (n^2 - 1)**.
- Conversely, if the branching factor is 100, the maze will expand in a diamond pattern, out from the seed. Collisions don't matter, because every seed stays a seed until it's completely surrounded. If we assume the worst-case scenario, where the seed is a corner cell, then the algorithm would require 2*(n-1) cycles to complete the adjacent edges, and another 2*(n-1) cycles to complete the remaining, opposite edges, for a total of **4*(n-1)** cycles.

Although these represent the **theoretical maximum and minimum** values, there is no substitute for experimental data.



BranchFactor	Avg	N^2	Min	Max
0	2886	2601	2177	3450
5	767	2601	653	966
10	463	2601	370	585
20	269	2601	257	301
30	234	2601	215	243
40	223	2601	215	241
50	224	2601	211	233
60	217	2601	213	223
70	214	2601	201	237
80	219	2601	201	255
90	201	2601	201	203
95	201	2601	201	201
100	201	2601	201	201

Observations

- As the branching factor approaches 50 (50% chance of a split), the run length approaches 4*n.
- At a branching factor of 0, the average run length is about **1.1 * n^2**.
- The min vs. max variance increases as a function of the average run length, inversely-proportional to the branching factor. At 0, there is a 44% variance, while there is only a 16% variance at 70.

Experimental Parameters

- Maze size: 51 x 51
- Fixed seed location: (0,0), which represents a worst-case scenario
- Tested over a range of branching factors, from 0 to 100.
- Each branching factor value was tested 10 times, to obtain min, max, and average values.

Produce a graphical, visual representation of the result set.

This capability has already been demonstrated multiple times throughout this document.

Implementation

- The proof of concept was implemented as a win32 binary executable.
- Images were saved in Windows BMP (Bitmap) format.

Real-World Applications

Gaming

Programmically-generated worlds are a huge component of modern game design.

Dynamic worlds offer replay value, and allows the developer to minimize the size of game assets by generating levels and entire worlds on the fly.

Customized Floor Plans

Everyone wants a custom home. Builders offer a variety of predefined floor plans, and offer limited customization.

Using a predefined rule set, builders could use a CA-based algorithm to offer an unlimited variety of floor plans.

Likewise, large facilities could be designed and optimized by CA algorithms.

Facility / Physical Security

Often, convolution is intentionally built in to the layout of secure facilities, and even gated communities, so that those who are unfamiliar with the layout will have difficulty with ingress and egress.

From an ingress perspective, convolution makes it difficult to locate a specific target. From an egress perspective, convolution slows a perpetrator down, giving security staff (who are more familiar with the layout) a chance to apprehend them.

Using a maze generator is an excellent way to introduce random convolution in to any layout.

Future Works

The following are potential project improvements:

- Adapt the algorithm to fill in irregularly-shaped areas. This would be based on a masking attribute, adding a “wall” (static) state for nodes that fall outside of the masked area. The remaining cells would be set to state 0 (disconnected).

- Generate a node graph of the finished maze. This would be especially useful for game design, as computer-controlled players (often called AI players, or “bots”) make extensive use of a node map for navigation, tactics, and goal-seeking.

- Have multiple seeds of different types, that run concurrently and compete for the same geography.

Conclusion

This document has demonstrated a practical method for generating and rendering random mazes using Cellular Automation.

Generating the maze is as follows:

- The maze starts with a single seed, which chooses an unused neighbor to invite, to become the new seed. A vector points to the invited neighbor.
- The seed switches to the ‘invitation’ state, which is a trigger to the invited neighbor to compare its position relative to the seed, to the invitation vector. If they match, the neighbor becomes a seed.
- The old seed, now in the ‘invitation’ state, either becomes ‘connected’ (stable), or within the branching probability, remains a seed, effectively creating a branch.
- If all seeds die, any ‘connected’ seed that neighbors an unused cell might, within the branching probability, become a new seed.
- This continues until there are no unused seeds.

Rendering the maze can be done in one of two ways:

- Draw nodes:
  - Start with an all-black bitmap
  - Draw each cell as a white block
  - Use the cell’s parent vector to draw a white block that connects the cell to its parent.
- Draw walls:
  - Start with an all-white bitmap
  - Assume each cell has 4 walls (N, S, E, W)
  - Use the cell’s parent vector to eliminate a wall
  - Examine each neighbor, and eliminate any walls that adjoin connected cells
  - Draw the remaining walls in black

Cellular automation is an excellent tool for programmatically-generating dynamic worlds and levels in games, and could also have potential real-world applications where differentiated designs and layouts can be generated using a base specification and a few simple rules.