Procedural Maze Level Generation with Evolutionary Cellular Automata

Chad Adams
University of Nevada Reno
Reno, NV 89557
Email:Adams.Chad7@gmail.com

Sushil Louis
University of Nevada Reno
Reno, NV 89557
Email:sushil@cse.unr.edu

***Abstract*—Maze running games represent a popular genre of video games and the design of playable mazes provides an interesting research challenge in procedural content generation for computational intelligence research in games. In this paper, we attack the problem of creating playable mazes by using genetic algorithms to evolve cellular automata rules that lead to playable mazes. More specifically, a fixed number of evolved-rule applications generates maze like patterns on a cellular automata grid and a region merging algorithm then generates the final, playable maze. Since maze path lengths correlate with maze playability, the genetic algorithm searches for cellular automata rules that lead to longer path lengths. Results from two types of cellular automata and three different fitness functions of path length show that our approach results in a variety of interesting, playable mazes with longer path lengths and complex paths.**

1. Introduction

Maze running games are a type of game where the player speeds through a maze to find an objective or to escape. The design of the maze makes these games fun to play and, traditionally, expensive game level designers handcrafted interesting and fun mazes. We attack the problem of automated or procedural maze generation by using a Genetic Algorithm (GA) to evolve cellular automata rules that lead to interesting mazes. Since "fun" and "interesting" are usually ill-defined, we had earlier used an interactive genetic algorithm where a human provided the fitness measure needed to drive evolution and thus generated subjectively fun and interesting mazes [1]. This prior work indicated that the length of the maze and the number of dead-ends were related to mazes classified as "interesting" by human players. Thus, in this work, we use and compare three simple fitness measures 1) the length of the maze, 2) the number of dead-ends, and 3) the sum of maze length and number of dead ends, to drive an elitist genetic algorithm towards generating interesting mazes.

The genetic algorithm evolves the rules of a two-dimensional, two-state Cellular Automaton (CA) which in turn generates the maze being evaluated. A two dimensional cellular automaton consists of a two-dimensional grid of cells and set of rules specifying cell-state transitions based on cells' neighbors. Given an initial state of cells on the grid, a cellular automaton computes by applying state transition rules to all cells on the grid thus leading to changes in cell state. Repeated rule applications lead to changing cell states until the grid stabilizes. Since the long term behavior of cellular automata are highly non-linear and the space of state transition rules can be prohibitively large, we use genetic algorithms to evolve rules that cause the CA to generate to maze like patterns on 2D grids. We then use a region merging algorithm [2] to generate interesting, playable mazes from this pattern. We define and compare deterministic and probabilistic cellular automata rules. Probabilistic rules differ in that they specify the probability that a cell state transition will occur.

Interesting mazes need to be neither too hard nor too easy to get through. A maze that is trivial, like the maze in Figure 1, does not pose a challenge to an adult player

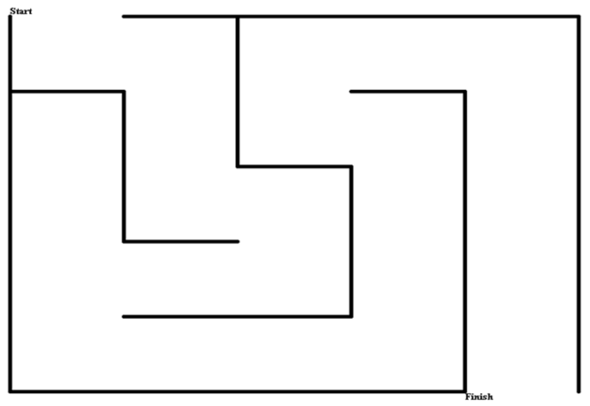


Figure 1. Simple Maze by by Mazes.ws [3]: not interesting due to being too easy

who will probably quickly lose interest. Conversely a maze that is too difficult to solve (see Figure 2) will probably tire and frustrate the player; again causing them to lose interest. Therefore maze design must carefully balance maze difficulty to keep a player both challenged and entertained.

Our preliminary results indicate that the genetic algorithm using the above three fitness measures generates

978-1-5386-2726-6/17/$31.00 ©2017 IEEE

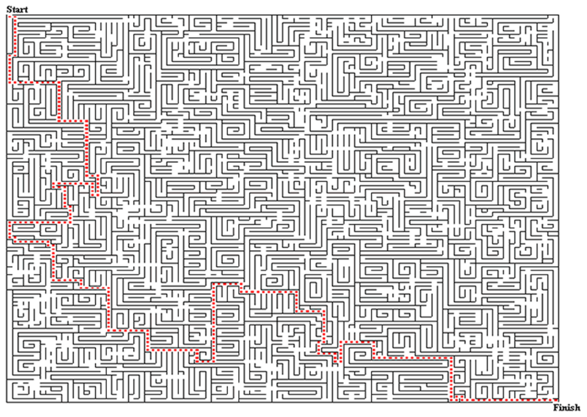


Figure 2. Difficult Maze by Mazes.ws [3]: not interesting due to being too difficult

mazes that are similar to those generated in our prior work where humans drove evolution by determining maze fitness [1]. That is, our genetic algorithm generates interesting mazes; mazes do not have very short solution paths, nor too few dead ends. Furthermore, comparing our GA generated probabilistic cellular automata mazes with mazes generated by Pech et. al., indicate that our generated mazes have a higher dead end count and fewer isolated filled cells [4]. Additionally, when comparing our probabilistic cellular automata mazes with mazes generated by D. Ashlock et. al. our generated mazes contained fewer solution paths, which was found to be related to how "interesting" human players rated mazes in prior work [1] [5]. These results show the potential of our approach towards maze generation and to help with level design in other game genres.

The next section describes prior work in this area. Section 3 defines our cellular automata and rule encoding in more detail, specifies the fitness function, and experimental methodology. Subsequently, Section 4 shows our results and compares them with prior maze generation work. The last section provides conclusions and directions for future work.

2. Background

Much work in the field of procedural content generation impacts maze generation. Procedural content generation refers to the concept of automated game level (or map) design. There are several mainstream games that create their content procedurally such as the famous Diablo game series [6], known for its procedurally generated maze-like dungeons. Platformer game level design is related to maze level design since platform levels contain maze-like components. Smith et. al. designed a level generator for a 2-D platformer game that uses a model of player action-rhythm to generate levels of appropriate difficulty [7]. Difficulty balancing being important for maze design made this work of particular interest. Compton and Mateas split levels into pieces, algorithmically generate the design on each piece and then combine the pieces together to generate a complete level for their platformer game [8].

Togelius et. al. provides a survey of contemporary methods of search-based content generation, genetic algorithms being a type of search-based approach. The paper defines several categories that specify the types of content generated, how to insure quality, and when content is being generated [9]. Insuring a contiguous path between the start and end of our generated mazes drew us to Togelius' work and assisted in our decision to incorporate region merging. Togelius and Schmidhuber present a search-based approach for procedurally creating an entire game in "An Experiment in Automatic Game Design" [10], where their evolution of the behaviour of all active elements in a game draws some similarity to our own work. Togelius et. al. present a multi-objective evolutionary approach for procedurally generating StarCraft maps [6] [11]. Togelius et. al.'s work shows similarity to our work in that it uses a GA to procedurally generate maps. Closer to our work in this paper we find Ashlock et. al.'s work [5] which explores a search-based approach for generating maze levels. They compare multiple methods for representing evolvable mazes. Of note for our work, their positive indirect representation functions similarly to our cellular automata approach. Their comparisons of differing fitness functions was of particular interest to our research where they noted that lengthening paths to cul-de-sacs did not produce mazes. They also use shortest path length, number of dead ends, and the combination of both. Closest to our work stands the paper by Pech et. al. who evolve cellular automata to generate maze levels [4]. Their work differs from ours in that they test evolving binary cellular automata rules dealing with between two (2) and four (4) cell states, what they refer to as flavors, while we test probabilistic rules on binary cell states. They also use a fitness function that measures similarity to a combination of nine target features, each with differing weights, while our approach only looks at shortest solution path length and number of dead ends.

Cellular automata were first invented by J. Von Neuman [12] with the most well known cellular automaton being J. Conway's Game of Life [13]. In Johnson et. al. cellular automata were used to generate infinite cave level maps [14]. The caves generated by Johnson et. al. share similarities in structure to mazes. R. Breukelaar and T. Bäck use a GA to evolve cellular automata rules in two dimensions [15] and multi-dimensional spaces [16]. Their results [15] show evolving CA rules as a viable route for procedural content generation by evolving two, three, and higher dimensional shapes.

The approach we use to attack the problem of maze generation differs from previous methods in several key ways. We use probabilistic cellular automata rules, use region merging to get rid of isolated regions, and we use a different method for detecting dead-ends. The next section details these differences.

3. Methodology

We use a genetic algorithm to evolve rules for a cellular automata (CA) in order to optimize three different fitness functions. These evolved rules are then applied to a maze of two separate starting states, a fully blank slate and where the center four cells begin as filled. After using the CA rules, a region merging algorithm is applied to ensure a fully connected maze, as CA's tend to create disconnected regions [2]. The fitness of the resultant maze would then be calculated, followed by crossover and mutation. Algorithm 1 defines our genetic algorithm. An individual maze is evaluated by

Algorithm 1 Maze Generation Genetic Algorithm

```
InitializePopulation();
EvaluatePopulation();
while $Generation < MaxGenerations$ do
    ElitistSelection();
    Crossover();
    Mutate();
    EvaluatePopulation();
    $Generation+ = 1$;
end while
```

Algorithm 2. Fifty (50) iterations usually suffices to stabilize cell states in the cellular automaton grid. We thus apply the rules of the cellular automaton on each cell and repeat this 50 times. We explain cellular automata in more detail in the next subsection. After the grid stabilizes (50 iterations), we run the region merging algorithm [2] to ensure that there exists a path from start to finish, and then measure fitness on the resulting maze.

Algorithm 2 Maze Evaluation

```
Iterator = 0;
while Iterator < 50 do
    ApplyCellularAutomataRules();
    Iterator += 1;
end while
MergeRegions();
GetMazeFitness();
```

3.1. Cellular Automata and Chromosome Representation

Cellular Automata describes a discrete model where cells on a regular grid change between a discreet number of states based upon the state of a defined set of cells called a neighborhood. We used a grid used with the dimensions of 30x30 , the full grid has the dimensions of 32x32 but the outer layer remains filled and not considered by the cellular automata. The standard CA uses a two dimensional grid, two discreet states, and a neighborhood for each cell of the surrounding eight cells known as the Moore Neighborhood [17], shown in Figure 3. Figure 4 shows an example CA running using a Moore Neighborhood and the rules shown in Algorithm 3. These are the rules to Conway's game of life [13].

We can track the progression of "Cell: i" in Figure 4 which begins filled in time step zero, becomes empty in time step one as "Cell: i" had only one filled neighbor in time step zero, and becomes filled again in time step

Algorithm 3 CA example rules

```
if Cell State == Empty then
    if Filled Cells in Neighborhood == 3 then
        Cell State = Filled;
    end if
else
    if Cell State == Filled then
        if Filled Cells in Neighborhood != 2 or 3 then
            Cell State = Empty;
        end if
    end if
end if
```

two due to having three filled neighbors in time step one. For this paper, we use the standard CA while focusing on the rule representation to alter the final grid state. A CA's rules are how a given cell will alter its state based upon the neighborhood of the cell. We use two discreet states (filled and empty) and the Moore Neighborhood, which means that each chromosome will have two rule sets, each rule set will have nine different rules, one for every number of possible filled neighbors zero through eight, as can be seen in Figure 5. A maze is generated from these rules by iterating through each cell on the grid and applying the rules from a given chromosome to them. Note that the cells do not update their state until after the CA rules have been applied to each cell on the grid. We are able to generate a maze this way by applying the CA rules from a chromosome 50 times. After applying the CA rules 50 times to the grid all disconnected empty regions become connected by running a region merging algorithm [2], which first detects each empty region of the maze that are separated from each other. The algorithm then picks the first two regions and empties the cells at the narrowest point of the divide before repeating the process until only one region remains. Shown in Figure 7 you can see the order that the region merging algorithm will detect each of the shown regions and that same order would be used to merge the regions together along the blue lines. The displayed order of region merging is because we follow a simple algorithm of starting at the lower left of the map and moving from left to right, bottom to top which then keeps the first shortest path found if there are multiple paths tied for shortest.

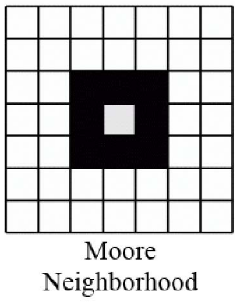


Figure 3. CA Neighborhoods

We encode our chromosome as an 18 bit array, nine bits

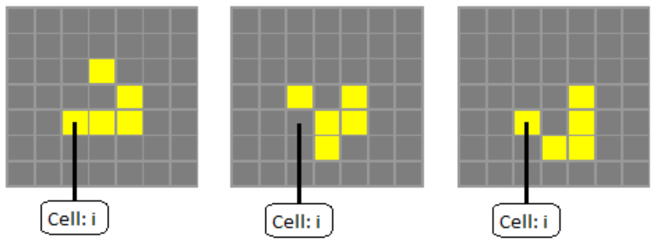


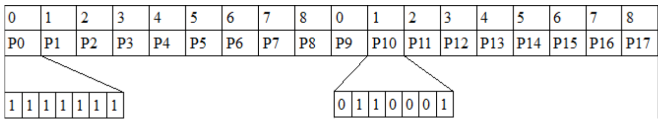
Figure 4. CA running: Left: Time step 0, Center: Time step 1, Right: Time step 2

for each cell state in our CA. Two chromosome representations were tested, the first was the binary representation and the second was the probabilistic representation. An example of the binary representation can be seen in Figure 5 where cells in the top array indicate the number of filled neighbors the corresponding cell in the bottom array relates to, with a one signifying that the corresponding number of filled neighbors will cause a given cell to become or stay filled and a zero indicating the cell will become or stay empty. The example rule set in Figure 5 will cause a given empty cell to become filled if it has two or three filled neighbors, or a given filled cell to become empty if it does not have two,three, or four filled neighbors. The binary chromosome representation, while one of the most basic cellular automaton rule sets, provides a good starting point to begin testing our approach for generating mazes.

0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0

Figure 5. Binary Chromosome format & example

The probabilistic representation has a similar structure to the binary representation, two discreet states of filled and empty and using the Moore Neighborhood [17], except rather than a binary "if x number of neighbors are filled then change state" each rule instead encodes a probability that a given cell will change state based upon the state of the neighborhood. Each rule can have a value from 0 to 127 which we convert into a seven bit number for ease of use in crossover for a GA. An example of the probabilistic chromosome representation can be seen in Figure 6, with the top row of cells representing the exact number of filled neighbors the corresponding cell in the bottom row relates to and the bottom row encodes the probability that the above number of filled neighbors will cause a given cell to change its state. The decoded values shown in the example chromosome of Figure 6 indicate that an empty cell has a 100% chance of changing state to filled if it has zero filled neighbors, while a filled cell has a 38.6% chance of changing state to empty if it has one filled neighbor. The encoded values are normalized to a value between zero and one during the CA rules application step for use as the probability the number of filled neighbors the value represents will cause a given cell to change state.



0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17

1	1	1	1	1	1	1

0	1	1	0	0	0	1

Figure 6. Probabilistic Chromosome format & example

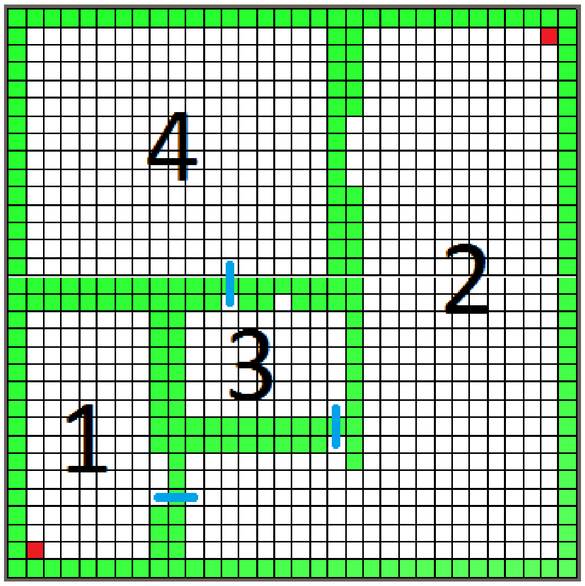


Figure 7. Region Merging Example.
Red: Starting/Ending Cell.
Blue: Path taken to merge regions

3.2. Genetic Algorithm

We use an elitist genetic algorithm where the top 'n' individuals from a population of size '$2n$' become the selection pool for crossover, producing n offspring, as well as being preserved into the next generation which produces the size $2n$ population of the following generation [18]. Since this selection is highly elitist, we use crossover rate of 100% and single point crossover to combat premature convergence and increase search space exploration. We use a mutation rate of 0.5%. These crossover and mutation rates were arrived at after some testing.

For the probabilistic chromosome our GA used a crossover rate of 90%, using half uniform crossover, and a mutation rate of 1%. Half uniform crossover crosses over half the differing bits [18]. These values were chosen to increase exploration and slow the rapid convergence rate that our elitist selection method produces. These values were also arrived at after some experimentation.

3.3. Fitness Evaluation

We tested three separate fitness functions and compared their performance against each other. The first fitness $(F1)$ function we used was 'Shortest Solution Path Length', the same fitness function from D. Ashlock's work [5]. We define 'Shortest Solution Path Length' to be the fewest number of contiguous empty map cells that connect the starting and end cell, including the end cell, with the fitness being the unweighted total length of the path. The shortest solution path fitness function was chosen as the longer the shortest solution path becomes the more ”interesting” the maze becomes [1]. The second fitness function ($F2$) used was ’Total Dead Ends’, with dead ends being defined as a map cell that has no neighboring cell with a longer path length to the entrance cell and the fitness being the unweighted count of dead ends. We chose this fitness function as a greater number of false paths makes a maze more interesting to solve, supported by results in our prior work [1]. Total dead ends, while similar to Ashlock et. al.’s work [5], differs from their fitness function in that we detect individual dead end cells instead of detecting grouped ’cul-de-sac’ cells at the end of a path. The third fitness function ($F3$) used was a linear sum of path length and dead end count without weight for either. We tested these fitness functions on two different starting map states. The first having the map start out as a Blank Slate where all cells start in the empty state which was chosen to minimize the impact that the starting map state had on the final maze layout. The second starting map state tested was where the center four cells started as filled while the remaining start as empty which was chosen to test if a minimally altered starting state would result in significantly different performance.

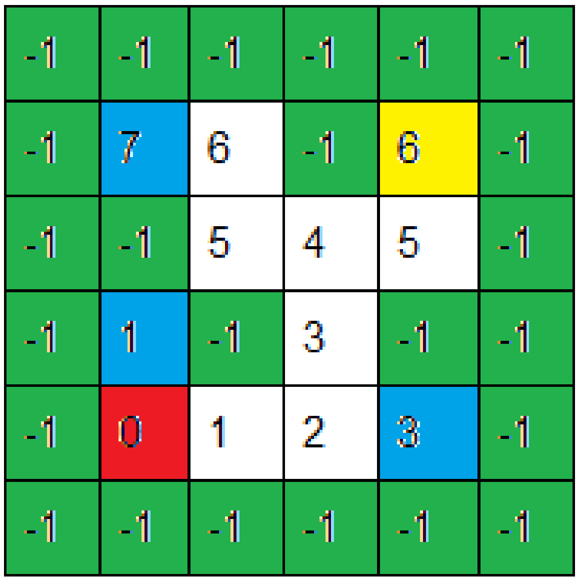


Figure 8. Fitness Evaluation Example.
Red: Starting Cell.
Blue: Dead End.
Yellow: Ending Cell

The path length of a given maze is calculated by running a recursive algorithm that marks each cell’s open neighbors with the given cell’s current shortest path length plus one if either the neighbor cell has not been visited or has a shortest path length greater than the value being assigned by the current cell. As our mazes are generated on a 30x30 grid, a given maze can have an absolute minimum path length of 58 as our starting cell has a path length of zero. Dead ends are detected by finding cells that have no unfilled neighbors with a greater path length value, with a check to ensure that the given cell does not fall directly between its unfilled neighbors in order to not mark hallways as having a dead end if both ends of the hallway are reachable by separate paths. Of note the ending cell counts as a dead end as it satisfies the criteria, meaning that any given maze has an absolute minimum number of dead ends of one.

4. Results and Discussion

We ran our GA 20 times for each experiment. In total we ran twelve experiments, testing each of our three fitness functions on both of our starting map states and chromosome representations. The first set of six experiments run were using our binary representation to set a good baseline performance level for our approach when expanded to larger search spaces. The second set of six experiments used our probabilistic representation.

4.1. Binary Representation

As our binary representation is exhaustively searchable (2^{18}), we were able to find the maximal performance for each of our fitness function by evaluating all possible combinations of 18 bits. Exhaustive search enables absolute performance evaluation and provides insight on expected performance when we go to larger search spaces such as with our probabilistic rule representation with a search space size of 2^{126}. Even with little tuning, our GA was able to evolve rules that consistently got within 85% of the optimum.

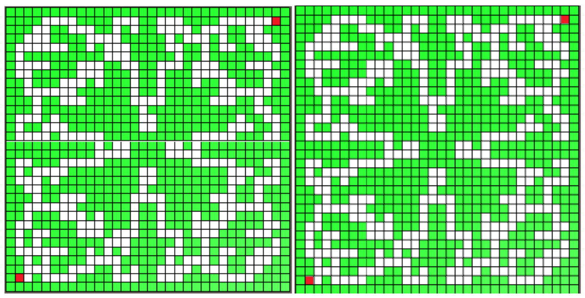


Figure 9.
$F1$ Binary Cellular Automata Exemplars. Left: Blank Slate. Right: Filled Center

Shown in Figures 9, 10, and 11 are the top performing mazes generated by our GA on both starting map states using our binary chromosome, the red squares indicating the starting point in the bottom left of the maze and the end point in the top right. $F1$ tends towards producing long snaking corridors with few branching paths. $F2$ produced some of the more interesting mazes, which tended to minimize the number of filled cells connected by a non-diagonal neighbor. $F3$ produced a combination of these two behaviours, with long snaking corridors with some sections

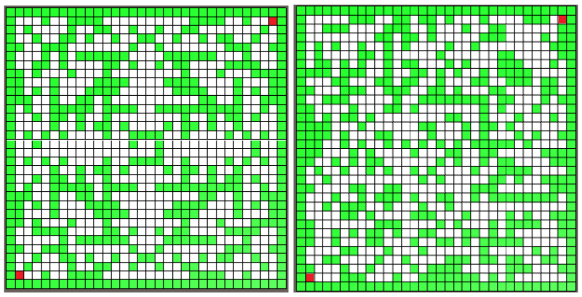


Figure 10.
$F2$ Binary Cellular Automata Exemplars. Left: Blank Slate. Right: Filled Center

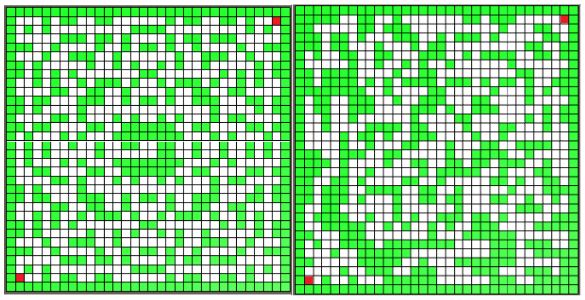


Figure 11.
$F3$ Binary Cellular Automata Exemplars. Left: Blank Slate. Right: Filled Center

connected only by a diagonal neighbor. The top row shows the mazes produced when the starting map state was empty. The mazes produced tended to be highly symmetrical, with a few minor differences that were created by the region merging algorithm.

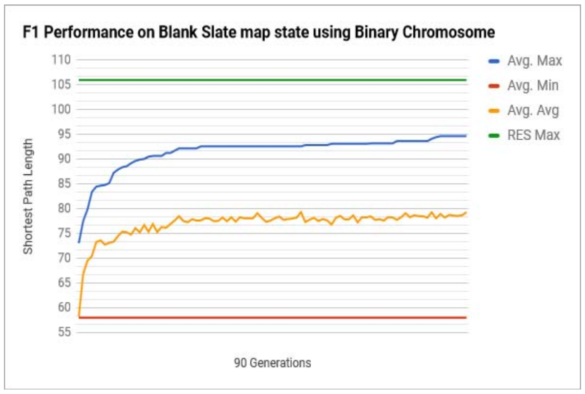


Figure 12. GA performance using $F1$ on the Blank Slate starting map state

Our GA was able to produce rules that on average produced mazes that were about 85% similar to the optimal solutions to our fitness functions as seen in Table 1.

Table 2 contains the average fitness values of our binary representation for each fitness function on each starting grid state. Shown in Figure 12 our GA's performance on $F1$ using the 'Blank Slate' starting map state can be clearly seen. Our maximum and average performance follows an

TABLE 1. FITNESS FUNCTION PERFORMANCE BY BINARY CHROMOSOME AS PERCENT OF MAXIMAL RESULT

Fitness Function	Blank Slate	Filled Center
F1: Path Length	89.3%	84.8%
F2: Dead End Count	88.6%	93.9%
F3: F1+F2	96.2%	89.5%

TABLE 2. FITNESS FUNCTION PERFORMANCE BY BINARY CHROMOSOME USING DIFFERENT STARTING MAP STATES

Fitness Function	Blank Slate	Filled Center
F1: Path Length	94.7	96.7
F2: Dead End Count	109.9	107.3
F3: F1+F2	175.9	184.4

TABLE 3. FITNESS FUNCTION PERFORMANCE BY PROBABILISTIC CHROMOSOME USING DIFFERENT STARTING MAP STATES

Fitness Function	Blank Slate	Filled Center
F1: Path Length	132.6	136.8
F2: Dead End Count	116.5	116.4
F3: F1+F2	219.4	218.5

expected trend. The average minimum performance tells a more interesting story, as it hovers around the minimum possible performance for the entire run. The cause of such a result could likely be due to the sensitivity of the cellular automata to mutations in certain rules, specifically in this case if the first bit mutates to a zero then the entire map will remain empty as no starting filled cells will appear. $F1$ has a similar performance curve on both blank slate and filled center starting map states while $F2$ and $F3$ follow a similar performance curve on both starting map states as shown in Figure 13.

4.2. Probabilistic Representation

Our probabilistic representation was able to perform better than our binary representation while maintaining similar improvement trends. The noted difference being that the average was able to converge to the maximum, which we infer as due to the representation being less sensitive to mutations of a single bit. The mazes generated by our probabilistic chromosome were able to further optimize our fitness functions as well as break from the symmetric patterns that the binary representation tended to create. The superior performance of the probabilistic representation over the binary representation can be seen by comparing the values in Table 2 with those in Table 3. The graphs in Figure 14 and Figure 15 show the performance curves for $F1$ and $F3$ respectively which follow similar curves both to one another and on both the blank slate and filled center starting map states, with the performance curve for $F2$ keeping with the same trend.

Figure 16 shows F1's performance with the probabilistic representation. The results tend towards long snaking paths with a few branching false paths, while these mazes do have paths that branch off from the solution path, they tend to be

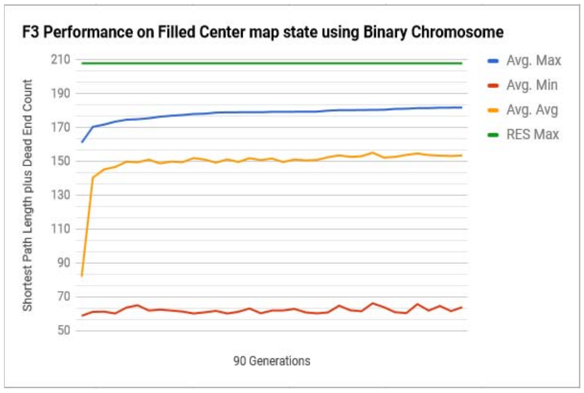


Figure 13. GA performance using $F3$ on the Filled Center starting map state

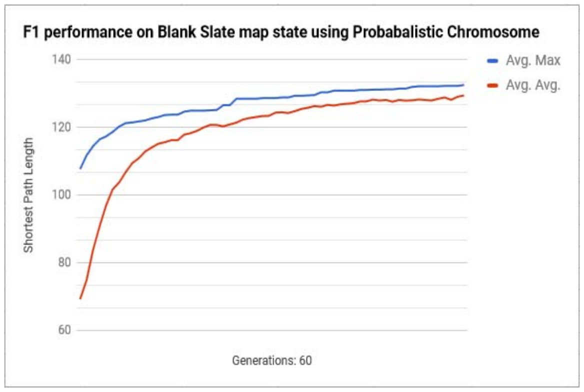


Figure 14. GA performance using $F1$ on the Blank Slate starting map state

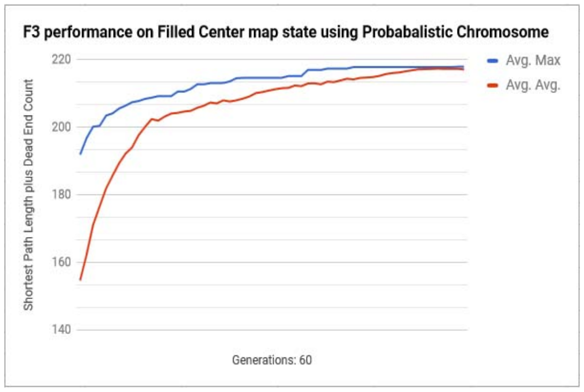


Figure 15. GA performance using $F3$ on the Filled Center starting map state

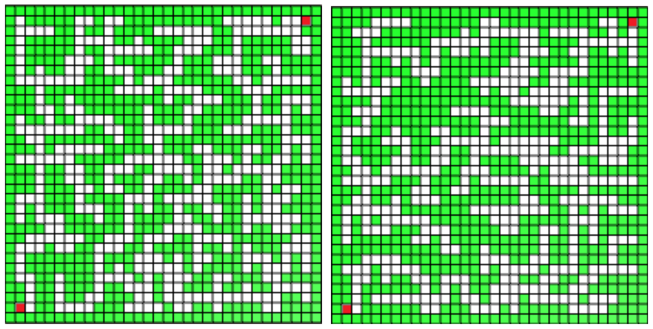


Figure 16.
$F1$ Probabilistic Cellular Automata Exemplars. Left: Blank Slate. Right: Filled Center

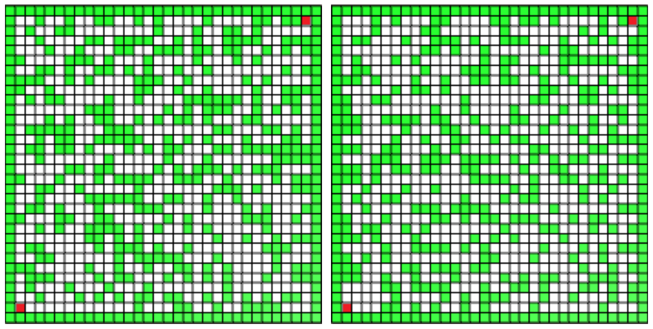


Figure 17.
$F2$ Probabilistic Cellular Automata Exemplars. Left: Blank Slate. Right: Filled Center

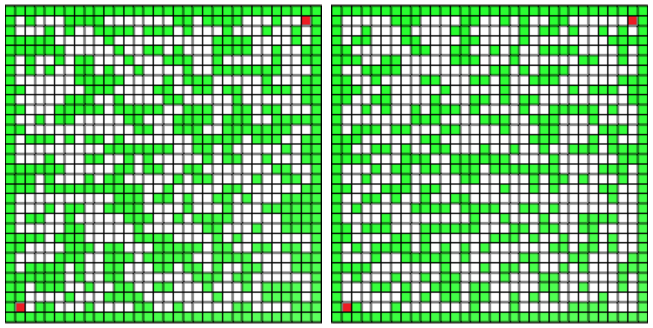


Figure 18.
$F3$ Probabilistic Cellular Automata Exemplars. Left: Blank Slate. Right: Filled Center

short. The mazes produced using F2 shown in Figure 17 have many dead ends, with more wall sections connected only to their diagonal neighbors, having long solution paths but frequently having more than one solution path. Mazes produced with F3 shown in Figure 18 combine the behaviours of F1 and F2 into one maze but with the notable traits of each being less pronounced, though F1's tendency to produce long snaking solution paths can be clearly seen having an effect on F3 in Figure 18. The most notable shared behaviour between all three fitness functions would be the tree-like structure produced in the mazes, where paths periodically branch off into new paths. The reason for this result we postulate as due to our method of region merging, which detects non-contiguous regions and connects them at their nearest point. Possibly randomizing the order in which regions are merged could produce different maze structures.

5. Conclusion & Future Work

Our research focuses on evolving mazes for a maze runner game using a genetic algorithm and cellular automata. The results show that our GA was able to evolve cellular automata rules that produced good maze levels with respect to our fitness functions. The superior performance of the probabilistic cellular automata leads us to infer it as a good possible avenue for future research. Cellular automata behaviour lends itself well to maze generation, and possibly to maps for other game types. The mazes that were generated by our approach have a higher number of dead ends, fewer solution paths, and less isolated filled cells, which is an improvement over previous approaches.

In the future we wish to utilize additional representations for cellular automata rules, such as having different rules for specific cell neighbors. Non-binary cell states, as used in Andrew Pech's work [4] and D. Ashlock's work [5], are an interesting possible future avenue for research using our fitness functions and region merging algorithm. R. Breckular's work [16] with higher dimensional cellular automata provides good incentive for expanding our approach to three dimensional grids in future experiments. Additional fitness functions such as distance from dead ends to nearest intersection or number of contiguous wall sections are another avenue for future exploration. A multi-objective fitness function would be an intriguing possibility due to the limited space that maze features can be placed in, for example the longer the solution path becomes then the shorter the incorrect paths must be and the less incorrect paths can be places, while still allowing for more fitness functions to be incorporated. An Interactive GA may also yield promising results when combined with our approach, either replacing or modifying the fitness functions, as well as getting a first person perspective for running the maze. The option of using our approach to generate maps for different game genres, such as Real Time Strategy games or First Person Shooters, presents another future research area we wish to explore.

Acknowledgments

This work is also partially supported by the Office of Naval Research under Grant N00014-15-1-2015

References

[1] C. Adams, H. Parekh, and S. J. Louis, "Procedural level design using an interactive cellular automata genetic algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 2017, pp. 85–86.

[2] S. Lague. (2015) Procedural cave generation tutorial. [Online]. Available: https://unity3d.com/learn/tutorials/projects/procedural-cave-generation-tutorial

[3] "Mazes.ws," http://www.mazes.ws/.

[4] A. Pech, P. Hingston, M. Masek, and C. P. Lam, "Evolving cellular automata for maze generation," in *Australasian Conference on Artificial Life and Computational Intelligence*. Springer, 2015, pp. 112–124.

[5] D. Ashlock, C. Lee, and C. McGuinness, "Search-based procedural generation of maze-like levels," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 260–273, 2011.

[6] Blizzard Entertainment, *Diablo*, 1997.

[7] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2d platformers," in *Proceedings of the 4th International Conference on Foundations of Digital Games*. ACM, 2009, pp. 175–182.

[8] K. Compton and M. Mateas, "Procedural level design for platform games." in *AIIDE*, 2006, pp. 109–111.

[9] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[10] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*. IEEE, 2008, pp. 111–118.

[11] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis, "Multiobjective exploration of the starcraft map space," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 2010, pp. 265–272.

[12] J. Von Neumann and A. W. Burks, *Theory of self-reproducing automata*. University of Illinois Press Urbana, 1996.

[13] M. Gardner, "Mathematical games: The fantastic combinations of john conways new solitaire game life," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.

[14] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 10.

[15] R. Breukelaar and T. Bäck, "Evolving transition rules for multi dimensional cellular automata," *Cellular Automata*, pp. 182–191, 2004.

[16] ——, "Using a genetic algorithm to evolve behavior in multi dimensional cellular automata: emergence of behavior," in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, 2005, pp. 107–114.

[17] E. F. Moore, "Machine models of self-reproduction," in *Proc. Symp. Appl. Math*, vol. 14, 1962, pp. 17–33.

[18] L. J. Eshelman, "The chc adaptive search algorithm: How to have safe search when engaging," *Foundations of Genetic Algorithms 1991 (FOGA 1)*, vol. 1, p. 265, 2014.