International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

# An Extensive Comparative Analysis on Different Maze Generation Algorithms

**Deepak Mane[1], Rajat Harne[2], Tanmay Pol[3], Rashmi Asthagi[4], Sandip Shine[5], Bhushan Zope[6]**

**Submitted**: 18/07/2023 **Revised**: 04/09/2023 **Accepted**: 25/09/2023

**Abstract**: In this paper, we compared maze generation algorithms. The nature of maze generation algorithms is analyzed, and the best amongst them is determined. The algorithms studied are Prim's algorithm, Kruskal's algorithm, DFS algorithm (Depth First Search), Ellers algorithm, Wilson's Algorithm, Hunt and Kill algorithm, and Aldous-Broder algorithm. The performance evaluation of the algorithms is determined by two parameters: the time taken by the algorithm to generate the maze and the space complexity of the maze. Evaluations are done based on the number of variables, including the number of intersections and dead ends visited, along with the overall steps taken by the agents, to determine how tough a maze is to navigate. The maze-generating algorithms are scored based on the performance of the agents. The algorithms' performance determines the most effective algorithms. The outcome is laid out analytically and graphically, showing a detailed analysis report of the algorithms' advantages and disadvantages. This report can be helpful in the field of gaming development, AI training, robotics, and automation. Such detailed comparative analyses have yet to be carried out on maze generation. Our research has bridged this research gap on maze generation algorithms and provided a detailed comparative analysis of maze generation algorithms.

***Keywords****: Artificial Intelligence, Maze Generation, robot, maze, and genetic algorithms.*

## 1. Introduction

Mazes are related to labyrinths, which have existed since antiquity. Typically, they are constructed using materials that are found in nature. They originally had a religious connotation [1]. Their significant goal later was amusement. Mazes have gained interest among scientists in more recent times, particularly mathematicians. Existing maze generation algorithms face several challenges that researchers and developers have been working to address. Current challenges include - Bias and predictability, scalability, path complexity, and dead ends.

Path-finding algorithms, robotics, and game creation are artificial intelligence disciplines that rely heavily on maze-generation methods.[8] There are many maze-creation algorithms, each with specific advantages and disadvantages. This paper intends to provide a thorough investigation and comparative analysis of numerous well-known artificial intelligence (AI) maze-generating systems. The algorithms' characteristics will be specifically examined to decide the best. Prim's algorithm, Kruskal's algorithm, DFS (Depth First Search), Ellers method, Wilson's Algorithm, Hunt and Kill algorithm, and Wilson's Algorithm are the algorithms examined in this study. The length of time it takes to generate the maze and the maze's spatial complexity will be used to rate each algorithm. Three path-finding agents that solve mazes have been created to evaluate the algorithms' effectiveness. These agents use DFS, BFS, and Dijkstra algorithms, respectively, and will be used to report the findings to evaluate and rank maze-creating methods. The data collected by these agents will be analyzed to determine which maze-generation algorithm is the most effective for various applications.

To perform a quantitative analysis of each algorithm to understand their technical and engineering aspects. Calculating and comparing each algorithm’s time and space complexity and assessing how well it performs when applied to difficult maze-generating challenges will be entailed. To achieve this, each algorithm will be programmed and tested on mazes of various sizes and difficulties. How long it takes for each method to generate a maze will be timed, and how intricate the mazes’ spatial layouts are will be kept track of. By performing simulations with the three agents developed for this project, the precision and effectiveness of each method in clearing the maze will also be assessed.

The evaluation of the efficiency and complexity of each algorithm is a crucial component of our research. It will be necessary to evaluate the trade-offs between time and

[1,2,3,4,5]*Vishwakarma Institute of Technology, Pune-411037, Maharashtra, India*
[5]*Symbiosis Institute of Technology, Lavale, Pune412115, Maharashtra, India*
[1]*dtmane@mail.com*
[2]*rajatharne@gmail.com*
[3]*tanmay.pol20@vit.edu*
[4]*rashmiashtagi@gmail.com*
[5]*sandeep.shinde@vit.edu*
[6]*bhushan.zope@hotmail.com*

space complexity and ascertain which algorithms offer the best compromise between the two. Performance optimization strategies for each algorithm, such as parallelization, pruning, and other tools, will also be examined.

The main objective of this study work is to offer insightful information on the most effective labyrinth generation algorithms for particular applications. The optimal solutions for various use cases can be found by comparing the scalability and performance of various algorithms. Our results can help algorithm developers create new and enhanced algorithms for artificial intelligence maze creation challenges that are more effective and efficient.

The majority of these algorithms' maze-creation applications have been documented, but there has yet to be a known comparative study for the stated issue. Section II describes the past studies done so far on maze generation algorithms. Section III described the methodology used for comparison through subsections - algorithms, flowcharts, and path-finding agents used. Section IV shows the detailed comparisons of all the algorithms used for generating mazes in the form of graphs, tables, etc., in the form of experiment results. Section V defines the conclusion of the survey done in this research paper.

## 2. Literature Review

The research focuses on a comparison of three algorithms - A* (A star), backtracking algorithm, and genetic algorithm (GAPP) -in order to find the quickest way across a maze. The algorithms are evaluated based on two criteria - path length and time taken to find the path. The study includes mazes of varying sizes and obstacle densities. The results of the study are presented both analytically and graphically. This literature review highlights the focus, methods, and key findings of the paper[2]. Several games employ maze generation to create maps, environments, and other graphical elements. They previously employed expanding tree algorithms, spanning tree techniques, and so on. However, predefined matrices were utilized and limitations were imposed on the design of the mazes by the techniques. A straightforward method for creating arbitrary-shaped mazes through the assembly of mazes is suggested in this study. The ultimate large labyrinth is likely to be a customised one because mazes can be combined in a specific way based on user desire. The technology can instantly create any new labyrinth if a database of mazes with players' playing histories or preferences is available. In addition, it may help players design fresh, challenging mazes for user-generated content.[3]

A. Flowchart

In this research, a novel genetic algorithm-based approach for creating video game levels is presented. Gene pool incorporates learning is the name of the suggested approach. This technique is useful in feature selection because it is broad enough to be used in a variety of game genres. This study looks for valuable patterns in certain training data and saves them all in a gene pool. The genetic algorithm is then used to discover the pattern combination that will produce the best results. The gene pool also maintains track of the quality of each gene to comprehend the most commonly observed pattern at various levels. In contrast to previous research, this study develops a novel testing game with complicated rules that are difficult to define using a simple 2D array. The outcome of this study demonstrates that the approach is capable of producing several intricate layers at once. In comparison to the dataset, levels created using this approach generally need roughly three times as many steps to solve[4].

In this study, they run the aforementioned program using some input text. For instance, Fig. 1 displays the outcome of executing text 1. The upper left corner of the maze indicated the starting point, while the bottom right corner indicated the ending location. One generation was meant to consist of 100 people. The tree-structured programmes' depth was specified to be between 1 and 8. There were 58 generations of the maze, with a generation period of around 7 seconds. The primary path's beginning and ending points and its length matched the provided information, allowing for the construction of the necessary maze. It is also possible to represent the typical curvature of the maze.It is thought that more individuals are required to build a labyrinth of higher size. We also discovered that some text input structures are, in theory, unable to fill the grid of the rectangular labyrinth completely[5]. This paper suggests building new maze representations using the DFS algorithm. This method fulfils the criteria for creating lengthy labyrinth routes. DFS can only create trees as a maze generator, making it possible to quickly use a variety of current algorithms used in game design, such as A*, to inspect the graph[6].

## 3. Proposed Architecture

Each algorithm’s time and space complexity will be checked individually and analyzed further. Later on, the best fit would be checked with our three AI agents. The proposed architecture consists of a score generator based on three parameters: time complexity, space complexity, and suitability with path-finding agents, as shown in Fig 1. below.

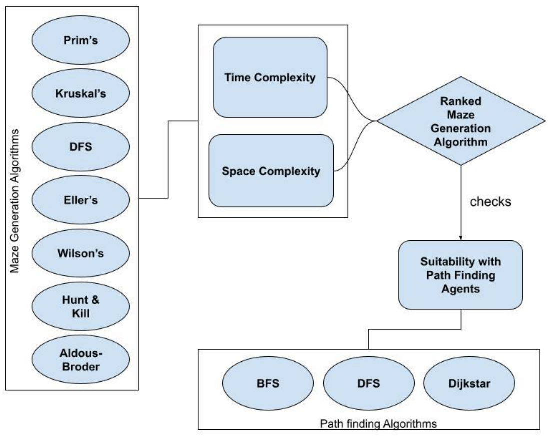

**Fig 1.** Flowchart of the ranking system

Fig 1. Shows, the ranking criteria for the maze Algorithms and then later these Maze generation algorithms are tested with the Path finding agents and analysis of their behaviour.

B. Algorithms

In figures 2 to 8, the green box is the source and the yellow box is the goal, while pink is the path of the algorithm and the orange box is the head of the path algorithms which would be looked into are as follows:

1. Prim's Generation Algorithm

a. Pick any vertex at random from G (the graph).

b. Two sets

i. Those who are already part of the maze's graph

ii. Those vertices (border) that are close to all of the related vertices

c. Select the edge in the frontier set with the least weight to connect to another labyrinth vertex.

d. Add the neighbours of the edge to the (border) and the edge to the minimum spanning tree (maze).

e. Points c and d would be in a single loop.

f. Visually expands forth from a location in the labyrinth.

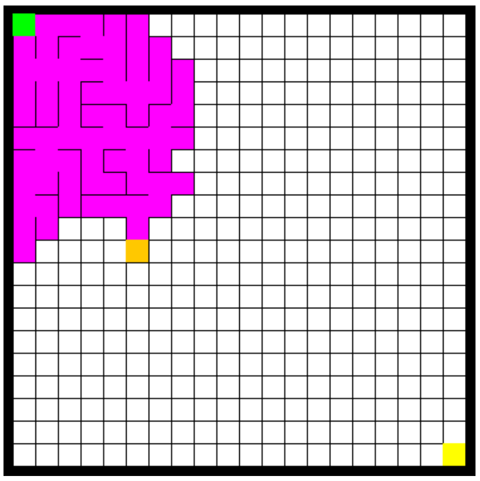

**Fig 2.** The initial stage of Prism's Algorithm

In Fig 2. Prism's Algorithm selects a random cell and then keeps searching adjacent cells to generate the maze.

2. Kruskal's algorithm

a. Pull all vertices into separate, disjoint sets starting with the graph's edges.

b. Select the edge with the lowest G weight. Join the trees together if the edge links two separate trees. Otherwise, discard that edge.

c. Repeat this until only one disconnected collection or tree remains.

d. Creates the labyrinth visually using numerous locations all across

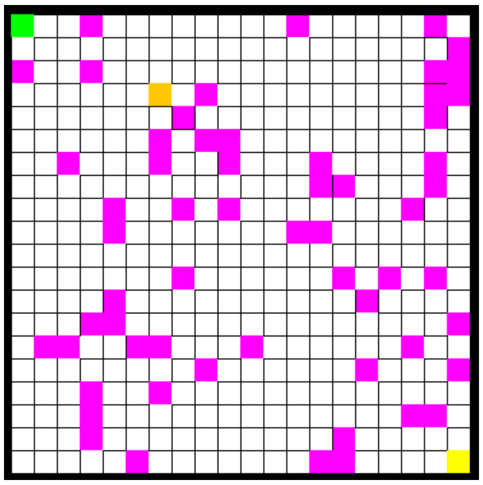

**Fig 3.** The initial stage of Kruskal's Algorithm

In Fig 3. Krusakal's Algorithm, a grid of cells and walls are first built via Kruskal's algorithm. When the cells on either side of the wall are in distinct sets, the wall is removed, the sets are merged, and the deleted wall is added to a list of removed walls. Next, walls are randomly selected. This procedure is repeated until all of the walls have been examined and taken down and there is just one set of cells remaining, which represents the finished maze.

2. Ellers algorithm

a. Initialise each cell in the first row to be in a separate disjoint set.

b. Only if they are not in the same set, randomly link neighbouring cells. Between the two joined cells, add an edge.

c. Randomly establish at least one downward vertical link and an edge for each set. Regardless of whatever set it is in, the cells in the following row must link to this one.

d. Create separate, disjoint sets for each of the remaining cells in the row.

e. Continue till the final row. Join all cells on the last row that don't belong to a set and don't have vertical connections that point downward.

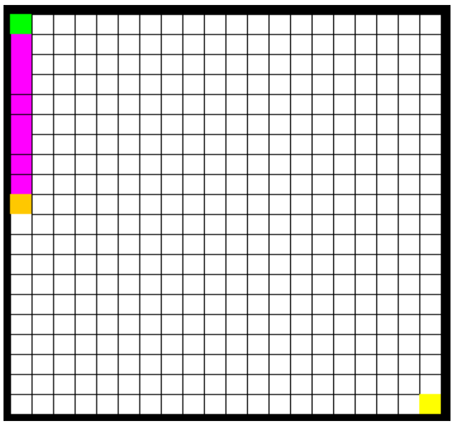

**Fig 4.** The initial stage of Eller's Algorithm

In Fig 4. Eller's Algorithm creates a single row of cells, each of which is in a separate set following that, the algorithm iterates through each cell in the row, choosing at random whether to connect the cell to its neighbour on the right or to create a new set for the cell. After that, the algorithm descends to the subsequent row and continues the procedure. Each set, however, has a chance to link up with the sets underneath it when advancing to the subsequent row. Until every cell can be reached from every other cell, the sets are connected at random.

3. Wilson's Algorithm

a. Select a vertex, then include it in the visited set.

b. Generate an edge (path) between the random walk and the visited set after executing a random walk on a randomly picked vertex until it encounters a vertex from the visited set.

c. If the random walk collides with itself, move the walk forward rather than creating an edge.

d. Continue till the visiting set is finished.

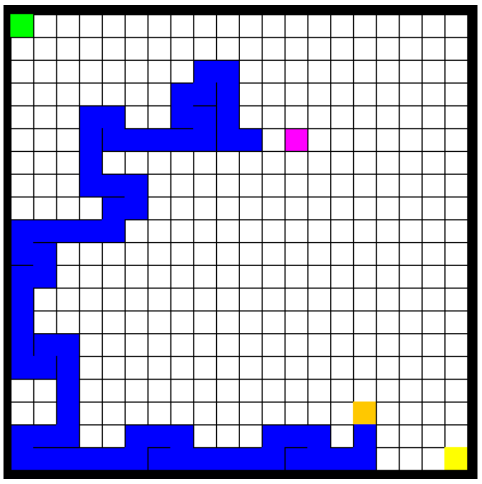

**Fig 5.** The initial stage of Wilson’s Algorithm

In Fig 5. Wilson’s Algorithm, it moves to an unvisited neighbouring cell that is at random from the current cell. Include the new cell in the random walk by marking it. The algorithm goes back along the random walk until it reaches the first cell that was visited if the new cell has already been visited. The method breaks down the barriers separating the cells in the random walk during this trace-back phase, essentially joining them. If the new cell has not yet been visited, the algorithm repeats steps 1-3 of the random walk starting from the new cell.

4. Hunt and Kill algorithm.

a. Identical to DFS, Picks a root vertex and proceeds to explore new vertices as far as feasible before turning around.

b. Switch to "Hunt" mode rather than going back. Find a cell that hasn't been visited that is close to one that has. Make a route (edge) connecting the two vertices.

c. Repeat up until there are no more unvisited cells.

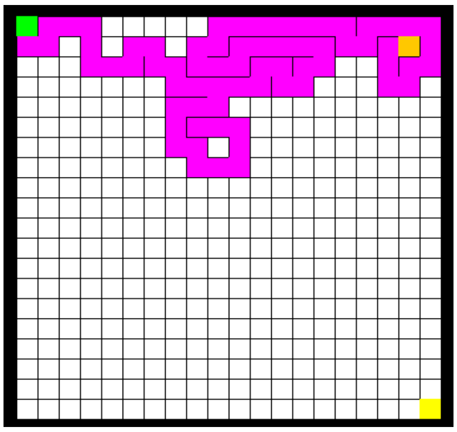

**Fig 6.** The initial stage of Hunt and Kill Algorithm

In Fig 6. the Hunt and Kill algorithm is shown creating a maze that begins with a single random cell and subsequently constructs the labyrinth by randomly chopping tunnels from visited cells to unvisited cells. The algorithm operates by continually going through two phases: a "hunt" phase in which it looks for an unvisited cell close to a visited cell, and a "kill" phase in which it creates a route to the unvisited cell and uses it as the new starting point for the hunt phase.

5. Aldous-Broder algorithm

a. Pick any vertex at random from G (the graph).

b. Go to a random vertex that is the current vertex's neighbour. Add the travelled edge to the spanning tree if the neighbour hasn't already been there.

c. Repeat the loop until each vertex has been reached.

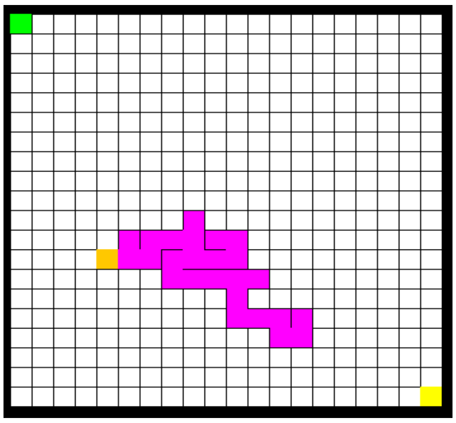

**Fig 7**. The initial stage of Aldous-Broder's Algorithm

In Fig 7. Aldous-Broder's Algorithm starts at a random cell in the grid. Choose a random neighbouring cell that has not yet been visited and move to it. If the chosen cell has not been visited before, carve a passage between the current cell and the chosen cell.

6. DFS algorithm (Depth First Search)

a. chooses a root vertex and proceeds to the maximum extent feasible before turning around.

b. A stack controls backtracking

c. Vertices are considered "new" if they have never been examined (or visited). Continually kept in a visited list

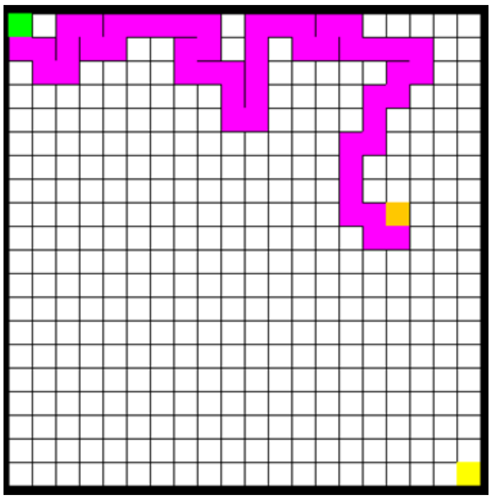

**Fig 8.** The initial stage of DFS's Algorithm

By observing Fig 8. in DFS, the algorithm starts at a randomly selected cell and moves on to haphazardly explore the neighbouring cells until it encounters a dead end, at which time it turns around and moves back to the last cell with unexplored neighbours.

C. Path Finding Agents

1. DFS Solver

Pathfinding in a graph may be accomplished using the graph traversal method known as DFS (Depth-First Search). The DFS algorithm visits all the nodes that may be reached from a beginning node given a graph and a starting node by travelling along all potential paths until it reaches the end node.[2]

The DFS algorithm for pathfinding is as follows:

a. Create a stack from scratch to hold the visited nodes.

b. Slide the first node up the stack.

c. Create a set from scratch to hold the visited nodes.

d. Pop the top node out of the stack while it is still full.

i. Mark the node as visited and add it to the set if it hasn't already been.

ii. Return the route if the node is the end node.

iii. Move all of the node's unvisited neighbours onto the stack.

e. Return "no path found" if the final node cannot be located.[10]

2. BFS Solver

A well-liked approach for determining the shortest path between two nodes in a network is called Breadth-First Search (BFS). The BFS method for path finding functions as follows:

a. Create a queue from scratch and add the initial node to it.

b. Create a visited set from scratch and include the initial node in it.

c. Dequeue the front node from the queue while it is still full.

i. Return the route if the front node is the goal node.

ii. Alternatively, for each of the front node's neighbours:

iii. Add the neighbour to the visited set and enqueue it if it hasn't already.

iv. Note the route taken from the origin node to the destination node.

d. There is no path if the objective node cannot be located.

e. The time complexity of BFS is O(V + E), where V is the count of nodes in the graph and E is the number of edges.

3. Dijkstar Solver

A well-liked approach for determining the shortest path between two nodes in a network is Dijkstra's algorithm. It operates by incrementally extending a search frontier from the beginning node, taking into account every nearby node and adjusting those nodes' estimated distances from the starting node as necessary. Until the destination node is reached or all reachable nodes have been visited, the algorithm keeps extending the frontier.

As long as there are no negative edge weights, this technique promises to locate the shortest route between any two accessible nodes in the graph. Use the Bellman-Ford method in place of the other algorithm if there are negative edge weights. [7]

## 4. Experiment Results

The complexity of the mazes created by the six aforementioned generating algorithms is examined in this section. The difficulty of mazes is assessed by examining their characteristics and the outcomes of solving agents. For completeness' sake, an experimental analysis of how well algorithms perform over time is also conducted.

A. Maze Generation Time Complexity

An experimental investigation of the time complexity of algorithms was conducted rather than a formal one. The focus was on the quality of the output that algorithms create, not on their flawless application. Hence, the study of temporal complexity was not given much attention.

How long maze-generating algorithms take to run is examined with respect to the size of the maze. The number of nodes in the maze, which would be a real metric, was not taken into account. Only the square grid graph's side length was utilised since the relationships between the performances of various algorithms are of interest. The final outcome is displayed in Figure 9.

In Table 1., Wilson's and Prim's algorithm takes more time in the 100x100 - 200x200 and 500x500-1000-1000 maze respectively and DFS take less time in every composition of the maze. In 1000x1000, around 2GB or less was consumed. We did not include 95+% memory consumption (10Gigabyte) in our observation since it would slow outcomes for 5,000x5,000.

| Grid | DFS | H&K | A-B | Wilson's | Prim's | Kruskal;'s | Eller's |
|---|---|---|---|---|---|---|---|
| 100x100 | 0.064 | 1.330 | 1.135 | 4.661 | 0.143 | 1.673 | 0.067 |
| 200x200 | 0.236 | 19.274 | 6.728 | 52.600 | 0.589 | 0.589 | 30.92 |
| 500x500 | 1.471 | NA | NA | NA | NA | NA | NA |
| 1000x1000 | 6.481 | NA | NA | NA | NA | NA | NA |

**Table 1:** Time Taken by Grid

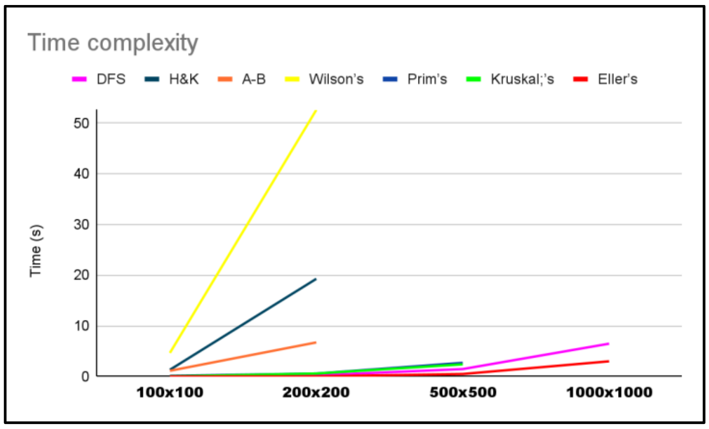

**Fig. 9.** Running time of the maze generating algorithms with respect to the maze size.

In terms of performance, certain algorithms stand out. They either move really slowly or a lot slower than anticipated: Kruskal is the sluggishest.

Wilson is quite ignorant and exhibits erratic behaviour, which slows things down once again.

The Aldous-Broder algorithm, on the other hand, is unexpectedly effective and fast. It is straightforward and makes use of primitives rather than advanced data structures.

## B. Maze Generation Complexity

The data structures used to describe the maze and monitor the algorithm's status determine the space complexity of maze creation algorithms. The space complexity, for instance, would be O(n2) if the labyrinth were represented as a 2D array, where n is the number of cells in the maze. This is necessary because we must keep a cell value (wall or passage) for every maze cell. However, some cells might need to store extra data, such as connection to other cells or visited status, which can complicate the available space. The Aldous-Broder method is most likely the best algorithm for labyrinth generation in terms of space complexity. The amount of memory required by this technique is independent of the size of the labyrinth since it has a space complexity of O(1) per cell. Using a random path through the maze, the Aldous-Broder algorithm visits each cell precisely once while creating a tunnel to a nearby, unvisited cell on each step. When every cell has been visited, the algorithm comes to an end.

The algorithm's space complexity is very minimal since it only has to keep data for the current cell and the preceding cell (i.e., two cells at most) at any one moment. Because of this, it produces huge mazes on systems with little memory, such as embedded systems or microcontrollers. The Aldous-Broder method may take a lot of steps to produce a labyrinth of decent quality, and it is important to keep in mind that it is not as time-efficient as some other algorithms. The selection of a maze-generating method is based on the requirements and limitations of the application, just like with any other algorithm. The Table 2. Given below represents the space complexity of each maze generation algorithm. Where n is the number of labyrinth cells and m is the number of graph edges. Few of the algorithms have space complexity as O(m+n) because these algorithms need to maintain a priority queue of edges and a set of visited cells. Aldous-Broder (A-B) has complexity O(1)per cell because this algorithm only needs to store information about the current cell and the previous cell at any given time, and does not depend on the size of the maze.

| | DFS | H&K | A-B | Wilson's | Prim's | Kruskal's | Eller's |
|---|---|---|---|---|---|---|---|
| Space Complexity | O(n) | O(n) | O(1) per cell | O(n+m) | O(n+m) | O(n+m) | O(n) |

**Table 2:** Space Complexity of all algorithms

In Table 3. Given below, the values given for each cell, represent the % for a normal maze generation. For a 100 x 100 Maze. These values show the complexity of each maze generated based on the number of dead ends, junctions, crossroads and Straight-aways present in the maze.

|  | DFS | H&K | A-B | Wilson's | Prim's | KrusKal's | Eller's |
|---|---|---|---|---|---|---|---|
| Dead-Ends% | 10.0 | 9.32 | 29.3 | 30.0 | 35.5 | 30.6 | 28.2 |
| Straight-Away % | 30.5 | 31.2 | 17.2 | 16.5 | 15.6 | 15.9 | 24.3 |
| Turns % | 49.6 | 50.2 | 27.6 | 27.2 | 19.5 | 26.9 | 22.5 |
| Junctions% | 9.67 | 9.29 | 22.1 | 22.4 | 23.4 | 22.8 | 21.6 |
| CrossRoads% | .172 | ~0 | 3.58 | 3.78 | 6.06 | 3.90 | 3.30 |

**Table 3:** Generation results

The following properties are considered to analyse the difficulty of a maze:

1. size *s*
2. number of intersections $n_i$ ; intersections are vertices with more than two neighbours.
3. a number of dead ends $n_{de}$; dead ends are vertices with only one neighbour.

The bigger $n_i$ the more difficult is the maze. The same goes for dead ends. One thousand mazes (of size 100 × 100) of each type were generated and the number of intersections and dead ends was calculated. The average results are shown in Table 4.

| Algorithms | $n_i$ | $n_{de}$ | Rank |
|---|---|---|---|
| Prim’s | 2937 | 3538 | 1 |
| Kruskal’s | 2657 | 3067 | 2 |
| Aldous-Broder | 2576 | 2979 | 3 |
| Wilson’s | 2574 | 2937 | 4 |
| Hunt and Kill | 926 | 934 | 5 |
| Eller’s | 768 | 793 | 6 |
| DFS | 647 | 659 | 7 |

**Table 4:** Average number of intersections and dead ends of the mazes.

The algorithms were ranked in accordance with the standards. The final difficulty ranking is shown in Table

5 as a result of assigning algorithm rankings for each metric and calculating the average of all the ranks.

| Rank | Algorithms |
|---|---|
| 1 | Aldous-Broder |
| 2 | Eller’s |
| 3 | Kruskal’s |
| 4 | Wilson’s |
| 5 | Prim’s |
| 6 | DFS |
| 7 | Hunt and Kill |

**Table 5:** Ranks of algorithms by the level of the difficulty.

## 5. Conclusion

Our research findings concluded on applying DFS to produce mazes quickly. If you need particularly large mazes (more than 1000 * 1000), choose a faster language like C/C++. In general, speed is frequently given up for organization while creating mazes. DFS was the fastest approach for solving mazes on average; simplicity is also not a negative thing for these size mazes. The more complicated the technique, the more interesting the mazes could be. Depending on the data structures you use in your maze and algorithms or other languages, you can obtain different results. This analysis would help in better training of path-finding bots. The future scope of our study would be to have Generative AI determine the fastest maze-generation algorithm trained on our researched data.

## References

[1] Gailand Macqueen. The spirituality of mazes andlabyrinths. Wood Lake Publishing Inc., 2005.

[2] M. Karova, I. Penev and N. Kalcheva, "Comparative analysis of algorithms to search for the shortest path in a maze," 2016 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom), Varna, Bulgaria, 2016, pp. 1-4, doi: 10.1109/BlackSeaCom.2016.7901597.

[3] -m. Bae, E. K. Kim, J. Lee, K. -J. Kim and J. -C. Na, "Generation of an arbitrary shaped large maze by assembling mazes," 2015 IEEE Conference on Computational Intelligence and Games (CIG), Tainan, Taiwan, 2015, pp. 538-539, doi: 10.1109/CIG.2015.7317901.

[4] K. Susanto, R. Fachruddin, M. I. Diputra, D. Herumurti and A. A. Yunanto, "Maze Generation Based on Difficulty using Genetic Algorithm with Gene Pool," 2020 International Seminar on Application for Technology of Information and Communication (iSemantic), Semarang, Indonesia, 2020, pp. 554-559, doi: 10.1109/iSemantic50169.2020.9234216.

[5] K. Okano and K. Matsuyama, "A Method for Generating Mazes with Length Constraint using Genetic Programming," 2020 Nicograph International (NicoInt), Tokyo, Japan, 2020, pp. 39-42, doi: 10.1109/NicoInt50878.2020.00014.

[6] Kozlova, J. A. Brown and E. Reading, "Examination of representational expression in maze generation algorithms," 2015 IEEE Conference on Computational Intelligence and Games (CIG), Tainan, Taiwan, 2015, pp. 532-533, doi: 10.1109/CIG.2015.7317902.

[7] K. Wei, Y. Gao, W. Zhang and S. Lin, "A Modified Dijkstra’s Algorithm for Solving the Problem of Finding the Maximum Load Path," 2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT), Kahului, HI, USA, 2019, pp. 10-13, doi: 10.1109/INFOCT.2019.8711024.

[8] X. He, Y. Wang and Y. Cao, "Researching on AI path-finding algorithm in the game development," 2012 International Symposium on Instrumentation & Measurement, Sensor Network and Automation (IMSNA), Sanya, China, 2012, pp. 484-486, doi: 10.1109/MSNA.2012.6324627.

[9] Jain, "Evacuation map generation using maze routing," 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT), Tiruchengode, India, 2013, pp. 1-6, doi: 10.1109/ICCCNT.2013.6726648.

[10] S. Hidayatullah, A. N. Jati and C. Setianingsih, "Realization of depth first search algorithm on line maze solver robot," 2017 International Conference on Control, Electronics, Renewable Energy and Communications (ICCREC), Yogyakarta, Indonesia, 2017, pp. 247-251, doi: 10.1109/ICCEREC.2017.8226690.

[11] Agnesia, and Wirawan Istiono, "Wall Pattern Detection with Prim�S Algorithm to Create Perfect Random Maze" Journal of Theoretical and Applied Information Technology 101, no.9.(2023)

.

[12] V. Bellot, M. Cautrès, J-M. Favreau, M. Gonzalez-Thauvin, P. Lafourcade, K. Le Cornec, B. Mosnier, S. Rivière-Wekstein, “How to generate perfect mazes?” Information Sciences, Volume 572, 2021, pp. 444-459

[13] Andrew Hernandez, Stephen Wright, Yosef Ben-David, Rodrigo Costa, David Botha. Optimizing Resource Allocation using Machine Learning in Decision Science. Kuwait Journal of Machine Learning, 2(3). Retrieved from http://kuwaitjournals.com/index.php/kjml/article/view/195

[14] Timande, S., Dhabliya, D. Designing multi-cloud server for scalable and secure sharing over web (2019) International Journal of Psychosocial Rehabilitation, 23 (5), pp. 835-841.