Performance and Artistic Quality of Path Finding Algorithms on Algorithmically Generated Mazes

Isaac Blaine-Sauer

December 2021

1 Abstract

Mazes have been a fascination of the Human race since the beginning of written history. There are examples of labyrinths and maze-like tombs throughout Greek mythology and carved into Egyptian pyramids. Mazes also are represented very heavily in the field of computer science with many of the fundamental algorithms taught in core curriculum able to be changed only slightly so that they create a maze like graph. Beyond their practical applications in deceiving grave robbers and as visualizations for algorithms, mazes also have an inate artistic value. Considering the rise of Artificial Intelligence in our everyday lives in everything from personal assistants to self-driving vehicles, it is only a matter of time before art is created by machines instead of people. Our goal was to create visually appealing mazes utilizing fundamental algorithms of computer science such that people would consider them as art. In order to find the optimal maze types for aesthetic appeal, we implemented multiple maze generation algorithms, pathfinding algorithms, and visualization techniques. We wrote two testing programs in order to gauge the difficulty and complexity of these mazes, and we created a survey to identify the preferences of typical people when it came to the appearance of the different mazes. We determined that the maze generating algorithms, Kruskal's and Sidewinder, would be the most effective mazes to market as art to the general public, due to their average time taken to solve and their visual appeal.

2 Problem Description

The problem we initially intended to solve was not so much a problem as it was an idea. As we investigate the usages of Artificial Intelligence in society today, we primarily think of it being used by tech giants and data companies trying to optimize ads to get people to buy more and more products. However, in reality, artificial intelligence is scattered throughout almost every sector on the planet. In business, artificial intelligence provides insights into spending, customer retention, best course of action, at a level not replicable by humans and at speeds hundreds of times faster. In medicine, artificial intelligence has created numerous advances in genetic sequencing and has even taken a physical form and performed surgery all on its own [8]. An argument can be made that Artificial Intelligence is not good for the health of our economy or the rate of employment. AI has replaced thousands of jobs over the fields of finance, transportation, and the service industry. [4] While some argue that this is the natural progression of technology and the next step in human development, it is hard to overlook the potential downsides that come with advancing artificial intelligence. One field of society that has yet to face the complete Artificial Intelligence takeover is the creative arts. Arts and culture have experienced significant economic setbacks from COVID-19 [7]. Across the spectrum of artistic and creative endeavors, restrictions on gatherings, changes in consumer behavior, and severe unemployment have taken a devastating toll on the sector [7]. Thankfully, with the recent rise in popularity of NFTS (non-fungible tokens), artists have gained a new platform to showcase their work and interact with even larger communities than before. Our thought process is that this emergence will catalyze programmers to utilize their skills to get into the art world. Going forward, we will describe how we plan to show the versatile applications of AI in the art world, opening up new opportunities to artists struggling in this technology driven world. Art is something that is so much more than just paint on a canvas or notes on a staff. It is about the feeling that it brings forth in the viewer or listener. This inspires the question of is it possible for us to create an AI that can understand the immense number of factors that go into the appreciation and creation of an artistic work? Our goal is not to create an artificial intelligence capable of creating art that mimics that of human creation, but simply to utilize the basic building blocks of machine learning and artificial intelligence, as well as path finding and maze generating algorithms, to automate the process of creating art successfully enough to where people find artistic value in what our algorithms have created.

While this seems relatively subjective to the perspective of the viewer, both my partner and I are (believe it or not) human beings who are well versed in art and music. By using some of the most essential algorithms in computer science, we will create mazes of many different varieties. The human race has had a fascination with mazes since the beginning of written history and so it seems a worthy start to computer generated art. Besides just visual value, we want our art to be engaging. So on top of implementing multiple maze generation algorithms, we will be applying path finding algorithms to each of the mazes in order to check for the difficulty of the solution. We do not want the individual experiencing the maze to be able to see the solution immediately. In order to make the mazes more appealing we will randomize the colors that they are generated with. As far as determining the artistic quality of the mazes, we will release a survey in order to get feedback on which mazes people enjoy the most.

3 Brief History

Over the past 70 years, computers have drastically changed human life. They have impacted almost every aspect of our lives from how to communicate to how we spend our free time. It is easy to see the profound effect computers have had on human history and we are still very much so at the beginning of the age of data and computing. One thing that has remained constant in this time of change has been our love of art and human expression. However, this raises the question; how long will it be until we rely on algorithms and artificial intelligence for the creation of art and music and will we be able to tell the difference? Because our project relies heavily on creating an output that is visually pleasing to people, I will be investigating what makes art appealing to humans and how to approach that when using code as a medium.

At a very basic level, art is an expression of human creativity. While some might argue that it should be appealing to the eye or the ear, there are plenty of forms of art that not only are not appealing but could be considered by some to be vulgar or disgusting. Given this, it might be wise to add to our definition. Art is an expression of human creativity meant to impart certain feelings or emotions to those who experience it. However, if we are to learn how we might be able to use algorithms to generate something that most people would accept as art, we need to break it down further. What are the components of art that draw the eye and keep the attention of the viewer? As humans we prefer something that makes us think, but is not complicated to the point of confusion. Some of the main factors that contribute to enjoyment of visual art are contrast, balance, symmetry, and areas of high visual salience. Some of the primary components that draw the eye are "contrast of luminance, curves, corners, and occlusions as well as color, edges, lines and orientation" [13]. When thinking about what kinds of visual stimuli have an abundance of corners, contrast, balance, and high salience, mazes or labyrinths come to mind quite quickly.

Mazes and labyrinths have been a part of human history for thousands of years. Whether it was the labyrinth that housed Theseus and the Minotaur, egyptian tombs used to dissuade grave robbers, hedge mazes in the estates of english aristocrats, or corn mazes that are set up just in time for halloween, mazes have always piqued the interest of the wealthy and affluent [14]. Considering one of the goals of our project is to create images that people will be willing to buy, having the basis of our art be one that has been a symbol of wealth throughout history will hopefully draw the attention of prospective buyers. In addition to this aspect of maze art, labyrinths have been used as a form of meditation that traces back to the Middle Ages [12]. Specific labyrinths are designed to be a meditative experience when either followed with a finger or physically moved through. If we can take inspiration from the shape of these mazes when designing our algorithm, we stand a greater chance of creating something that represents human-made artwork.

One downside to using mazes as our art is that mazes are quite abstract and are quite obviously generated via computer. Our job as designers is to add enough features to stimulate what interests people in art while still maintaining the integrity of creating computer generated art. In order to actually generate our mazes we will need to consider many different maze generation algorithms. While we will initially want all of our mazes to be perfect mazes, i.e. a maze that has only one solution that can be accessed from any point, eventually it would be interesting to allow for imperfect mazes to see how our pathfinding algorithms manage. Because we will be overlapping multiple shapes, we will not be generating mazes in perfect rectangular shapes. This means we will have to modify our algorithms so that we have dead cells which can not be used in the maze. This can be done by masking then killing cells by removing certain cells from their neighbors list of neighbors [2]. Another important aspect of maze generation is difficulty.

Some factors that contribute to the difficulty of a maze are its size, number of intersections, and number of dead ends [5]. Based on these attributes, it seems as though Prim's algorithm should be the most difficult for a path finding algorithm to solve, however in experiment results, it was found that Aldous-Broder was the more difficult in terms of time taken to solve the maze and number of wrong turns taken. This was followed by Wilson and Kruskal, with Recursive backtracking being the least difficult. The level of difficulty is influenced by certain biases that are present when generating the maze. Aldous-Broder has no bias making it particularly difficult to navigate [5]. One final thing that we will want to consider when choosing which algorithms to implement is the visual appearance of the mazes.

Each of the algorithms mentioned above produces a maze that differs in appearance from the others, some of them more so than others. Considering contrast between elements of art is something that humans find appealing, it would be in our best interest to select algorithms that differ most in appearance from each other, while still maintaining a high level of difficulty. While a binary tree generated maze is quite easy to solve because of the major bias towards the lower right corner [9], it provides an interesting contrast to other mazes because of its structure. This would be great if it were implemented with a maze such as Kruskals or Prims considering how Kruskal's often has a large amount of dead ends and short passages vs the long biased passages in binary tree generation. Another algorithm that would be able to stand out between the other two is recursive division. While it is also a very low difficulty maze, it has a unique appearance where there are plenty of long straight walls giving it a segmented look [9].

There are many methods for solving a maze which we will touch on later. For the time being, we will focus on procedurally generating mazes. In many cases, mazes are created manually, but this requires many hours of practice to design paths with complicated structures. Luckily, there is no shortage of computer algorithms for drawing mazes. In a paper published by Peter Gabroski, we see the use of many maze-generating algorithms such as Prims, Recursive Backtracking, Aldous-Broder, and many more [6]. These algorithms all take different approaches, such as following one path to the end then backtracking if it does not yield a solution path. While the list of algorithms mentioned in the paper is extremely valuable and effective, there are many that bring maze generation to the next level. Consider a maze with many possible exit points and therefore many possible paths. In a paper published by Paul Hyunjin Kim, we see the use of an algorithm that leaves a number of exits specified by N [10]. In this case, there will be multiple solutions. This also opens up the possibility of multiple cost-optimal paths, in which case we could find ourselves in a tie. Since giving the tie to either solution wouldn't mean much given the cost will stay the same, a tie can be randomly decided or go to the first cost-optimal path found.

Here we find ourselves in the meat of this project, choosing and implementing maze-solving algorithms. The goal of this stage of the process is to evaluate the performance of different searching algorithms in terms of finding the optimal solution, runtime, and memory usage. In an article published in Scientific Research, A* search is applied to images of mazes and maps to determine the shortest path from a start point to an endpoint [1]. The main difference here is that the project under discussion applies this algorithm to images taken from different angles. Regardless of this, the implementation of A* will remain the same since the image is processed into a maze before any algorithm being applied to it. The writers go on to discuss the versatility and efficiency of A*, particularly in the context of games where characters are constantly moving. In a part of the Emergence, Complexity, and Computation book series, researchers explore several chemistry-based concepts for maze solving in 2D standard mazes which rely on surface tension-driven phenomena at the air-liquid interface. They show that maze solving can be implemented by using active droplets and/or passive particles [**?**]. This is a very interesting concept and while the applicability of this to our project would change it fundamentally, it is something we could explore in another iteration. Creating a video of an active droplet solving a maze independently would be an interesting visualization and could serve as a great NFT. In addition to its function as an NFT, we could go a step further and compare the performance of the active droplets with various search algorithms such as A*, BFS, and DFS.

In many of the publications we have looked at, there is a lack of informative conclusions related to how we wish to present our findings. For example, in an article by Navin Kumar, the performance of each search algorithm is indicated by labeling it Low, Medium, High, or Very High [11]. In terms of reviewing the performance of algorithms, we will take a more technical approach, by exploring how to track the runtime and memory usage of each algorithm. Before moving on, what if we decided to use memory-bounded algorithms? In a paper published in the FLAIRS conference, we see the implementation of a memory-bounded A* search [15]. While it is typically known how this will work compared to A*, it would be interesting to apply it to mazes with multiple exits and also to see the visualization of such a search algorithm as it prunes nodes and traverses the maze. Now that we have had the chance to go in-depth into the work done on maze-solving algorithms, we will switch our focus to the work done on image generation.

After a maze has been generated and solved, all the data will exist in arrays, so how do we go about visualizing it? There are a lot of options available to developers for maze visualization. A few notable ones are OpenCV, MatplotLib, Scikit-Image, and SciPy. In an article posted to LVNGD, we see the visualization of a maze by way of MatplotLib [3]. In our project, this is one of the most important aspects due to the end-goal which is having viewable “assets” that we will then upload to an NFT exchange. Undoubtedly, there will be a number of different iterations for the final form of the NFT. MatplotLib is a great choice for such development because they offer many choices for different design techniques such as 2D and 3D image generation. There are quite a few attributes we need to consider while designing our algorithm. Difficulty, artistic quality, appeal to the eye, and time of generation are just a few of those. In the end we hope to end up with a product that not only is able to be solved by pathfinding algorithms, but is appealing enough to be bought as an NFT.

4 Design Process

In order to approach this problem, we initially had to decide on the scope we were looking for this project. With this being the first time either of us had forayed into computer generated art, we decided it would be best to start by implementing the base features and to add more complexity as we went. The very first thing we had to decide was how we were going to represent our mazes. We had a couple of options like utilizing numpy or OpenCV for not only visualization but for the storage of the maze data. Due to difficulties with OpenCV, we decided to use a cells and grid implementation in Python. Each cell represented a singular square in the maze and consisted of two coordinates, (x,y), indicating its place in the maze and a dictionary of walls which listed whether the walls to the north, south, east and west were present or not. At its most basic form, our grid class consisted of a map of cells in a two dimensional array. From there we needed to write our first generational algorithm. This required researching how different maze generating algorithms affected the visual aspects and complexity of the mazes they created. Started with the binary tree generation algorithm in order to have a maze we could begin working on visualization with. The binary tree algorithm is quite simple and adheres to the following principles. For each cell in the map, randomly connect with either their north or west neighbor, assuming they exist. The mazes generated by the binary tree algorithm have a distinct look with most of the passages going from top left to bottom right. In order to actually visualize the maze however, it was necessary for us to create some sort of output that resembled a maze. We decided to use an ASCII based representation at first which, while crude, would allow us to see if the maze was being generated properly. An example of this can be seen in Figure 1.

```
+-+-+-+-+-+-+-+-+-+-+
|                   |
+ + + +-+ + +-+-+ + +
| | |   | |     | | |
+ + +-+ +-+-+ +-+ +-+
| |   |     |   |   |
+ +-+ +-+-+ + + + +-+
|   |     | | | |   |
+ +-+ +-+-+-+-+-+-+-+
|   |               |
+ +-+-+-+ + + + + + +
|       | | | | | | |
+ +-+ + +-+-+ + + + +
|   | |     | | | | |
+ +-+-+-+-+ +-+ + +-+
|         |   | |   |
+ + + +-+ +-+-+-+ + +
| | |   |       | | |
+ +-+-+-+-+-+ +-+ + +
|           |   | | |
+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Example of a 10x10 binary tree maze represented in ASCII

Once we determined that our mazes were generating properly, we began working on implementing more algorithms. The algorithms became more complicated to implement as we continued. The next maze generation algorithm we coded was the sidewinder algorithm, which is similar in essence to the binary tree algorithm. It follows the subsequent steps: For each cell in the grid, randomly choose if you are going to carve a passage to the east. If you choose to carve the passage, add the cell to the current set. If you do not carve the passage, randomly choose one of the cells in the run set and carve a passage north, then empty the run set. This maze has a look that can be changed based on the probability of traveling east that you set. For our mazes we set the probability of carving a passage east to 1/3 in order to give the maze a more waterfall-like appearance.

We implemented two more maze generating algorithms at this point, depth-first and kruskal's. Both of these algorithms have radically different implementations and also provide starkly different looking mazes from any of the other three algorithms. Depth-first was rather simple to code considering it follows a lot of the same principles as depth-first search. The steps are as follows: Choose any starting cell in the maze and add it to a stack. While there are any cells in the stack, pop the top cell and connect with all of its neighbors. You will then choose one of the neighbors to be the new current cell and add the rest to the stack. This creates a singular long path that follows its way from the start node to the finish with lots of smaller intermediate paths that stick out from it. It has no apparent bias and creates some mazes with a long run value.

```
+-+-+-+-+-+-+-+-+-+-+-+-+
|*|*****| | |   |       |
+*+*+-+*+ + +-+ + +-+-+-+
|*|*|  *|     |   |     |
+*+*+-+*+ +-+-+ +-+ +-+ +
|***|  ***|     |   | | |
+ +-+ +-+*+ +-+ +-+ + + +
| | |   |* |         | |
+-+ +-+-+*+ + +-+ +-+-+ +
|   |   |*| |   |   |   |
+ + +-+ +*+ +-+-+-+ + +-+
| | |*****| |   |   |   |
+ +-+*+-+-+-+ +-+-+-+-+-+
|   |*****| | | |   | | |
+ + +-+-+*+ + + +-+ + + +
| | | | |*|   |         |
+ + + + +*+ +-+-+ +-+-+-+
| |      ***|***  | |   |
+ +-+-+ +-+*+*+*+-+ + + +
| |     | |*|*|*****| | |
+-+-+-+ + +*+*+ +-+*+ +-+
|       | |*|*| |  *****|
+ + +-+ + +*+*+ +-+ + +*+
| |   | |  ***| |   | |*|
+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: Example of an ASCII 12x12 Maze Generated by Kruskal's Algorithm and Solved with DFS

Finally, we implemented kruskal's which, at the beginning of the project, we were most interested in because it creates very complicated mazes. However, it was by far the most difficult to implement as well. It required a very different approach from any of the other algorithms considering it requires a set of walls as well as a set of cells. The implementation was as follows: Create a distinct set for each cell in the grid and a set of all of the edges for all of the cells. Take an edge from the set at random. If that edge connects two disjoint sets of cells, connect the cells at the edge and join the subsets. Due to the way that we originally created our cell class, grid class, and get neighbor function, it was difficult to set up the original set of edges. However, this algorithm creates the most complicated mazes and gives a texture of many short paths and intersections.

Once we had all of the maze generation algorithms complete, we moved on to being able to actually solve the mazes to assess the difficulty. There are a couple of different ways to implement these algorithms, however we decided that the most versatile course of action would be to represent the mazes in the form of a graph. To do this, we first converted the maze array into an adjacency matrix, then we converted that into an adjacency list. Having the maze in the form of an adjacency list allowed us to access a broader scope of maze solving algorithms. Based on the fact that we have a completely unweighted graph, we decided the following three algorithms would be best suited to test the mazes: Breadth-first search, depth-first search, and A* search. Breadth-first and depth-first search are very similar in their implementation aside from which end of the queue you get the next cell. In breadth-first search, we search each sibling node (i.e. each neighbor of the current node) before expanding along any of the children paths. However, with depth-first search, we pick a single node to expand upon while adding all of the neighbors to the end of the queue.

For A* search, we had to take a different approach. Because A* takes a search heuristic as one of the primary parts of its algorithm, we had to decide on which heuristic to use. This was made more difficult by the fact that we were using an unweighted graph. We decided to use Manhattan distance as it would allow for us to check nodes that move closer to the goal first before checking any others. These algorithms were checked by outputting a path to the terminal in the form of an array, however we wanted to be able to see the solutions on the actual maze. So we incorporated a function that changed the ASCII representation so that the solution was visible in the form of a line. This can be seen in Figure 2. While this representation of the maze showed what we needed while testing our algorithms to make sure they worked, it was not something that most people would consider artistic in the slightest. In order to display the mazes in a more attractive way we chose to utilize numpy and matplotlib. This involved learning about the different ways that figures can be created and presented in matplotlib. Eventually we settled on using the 'pcolormesh' function which takes in a two dimensional array and represents it where each pixel is colored based on the corresponding value in the array. We were able to use colormaps to choose how to color the mazes. We then randomized this so that with each maze we generated, we were able to get a unique color combination on top of an already completely unique maze. An example of one of these representations can be seen in Figure 3. At this point, we created a way for the path to be visible on the plot in addition to the ASCII representation. We changed the array that we fed into our plot function so that the path represented by '*' in the ASCII code would be represented as the number two in the array. This gave us a way to create visuals such as the one seen in Figure 7

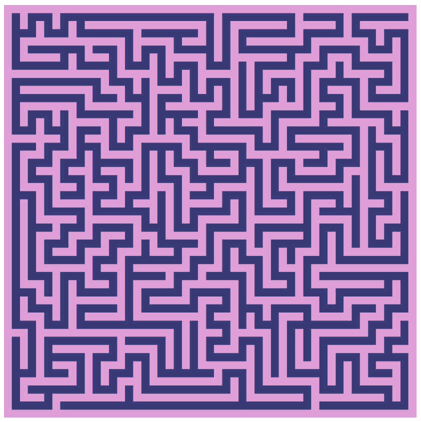


Figure 3: Example of an 25x25 Maze Generated by Depth First Algorithm. Visualized with Matplotlib

5 Experiment

Our problem was multifaceted so it was necessary to design multiple tests in order to quantify the success of our work. On one hand we are looking for tests involving the difficulty of the mazes and the time it took to complete for each of our pathfinding algorithms. We also need to find out which mazes are most visually appealing to the average person. In order to test the algorithms we will have to make use of the time module of python and write a program that runs each of the algorithms a certain number of times on differently generated mazes and mazes of various sizes. While solution speed is certainly important, another crucial aspect is seeing how the algorithms perform when the goal is not set to the opposite corner of the maze. For instance, we could hypothesize that simply due to the nature of how binary tree mazes generate, they might be solved very quickly if the goal node is in the bottom right corner. However, if we instead test for the solution being in the middle of the bottom compared to its original position, we could get very different search times. Another important factor to consider besides solution time is the length of the path to the solution. The longer the path to the solution, not only the more difficult a maze could be considered, but also the more time it will engage a viewer for.

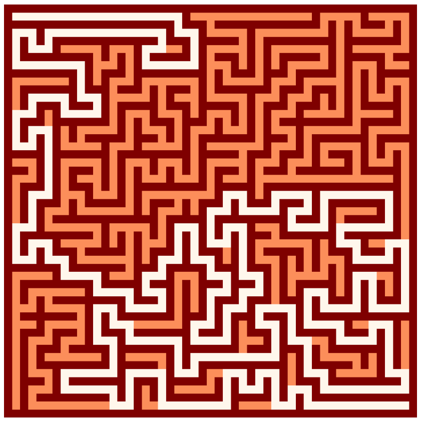


Figure 4: Example of an 25x25 Maze Generated by Depth First Algorithm. Visualized with Matplotlib and Solved with A* search

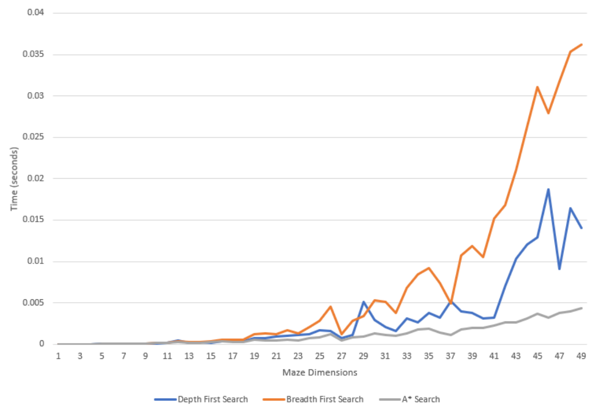
In order to obtain data for both of these statistics, we wrote two testing programs. The first of which had the purpose of measuring solely the time taken for each algorithm to solve each of the different types of mazes as they increased in size and complexity. We started with a 1x1 maze and ran each pathfinding algorithm 5 times before incrementing the size by one all the way up to a 49x49 maze. We timed each of the individual runs of the algorithm and stored all of the data in a two dimensional array. We then utilized the csv module in python to export it to Excel for our analysis. We repeated this process for each of the four maze generation algorithms.

For the second test, where we were looking for how the algorithms would perform when we changed the goal location, we were able to reuse some parts of our original testing program. Most of the logic behind testing the algorithms were the same, it was just necessary for us to run each algorithm twice where we changed where the goal was located for the second run. Because we wanted to have a visible difference in solution time, we started at a maze size of 10x10 and went up to a size of 25x25. We tested each of the algorithms against each other on every size and ran the tests five times. Using the same csv module we exported the data to Excel for analysis.

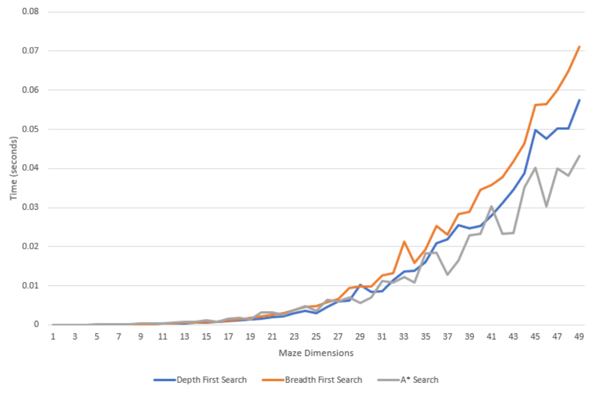
While these two tests helped us to determine the differences between our algorithms in terms of difficulty, it did not give us any data on how people interpreted the artistic value of our mazes. For this we created a google forms survey with pictures of mazes. We asked the participants to choose which maze they found most interesting and visually appealing. In each of the questions we changed only one variable such as color scheme, maze generation technique or size. We sent this out to our friends and family and posted it to online forums. We also asked the participants if they would be willing to pay money for the maze if it were art in the way it was currently and if they would pay money if they were more complicated or layered in some way. We then exported this data to Excel for analysis. The survey can be found at Maze Art Survey.

6 Results and Analysis

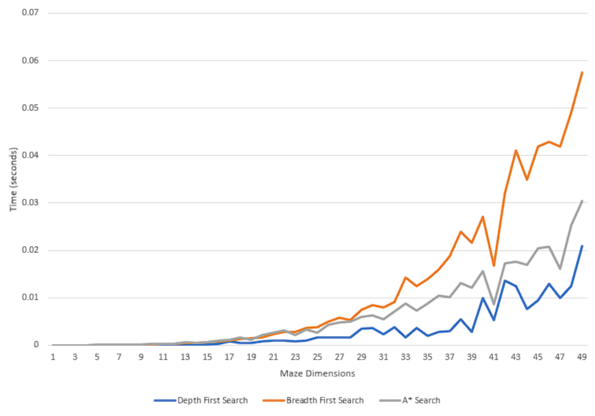
To analyze the results, we imported all of our data into excel. Because we ran each of our tests 5 times, we averaged the data from each of the runs and used this to plot Figures 5 and 6. Looking at Figure 5 we can see that for all of our maze generation algorithms, breadth-first search took considerably longer to find the solution than either of the other two algorithms. This makes logical sense because when breadth-first search runs, it checks each neighbor of the current cell before moving on to the next cell and moves through



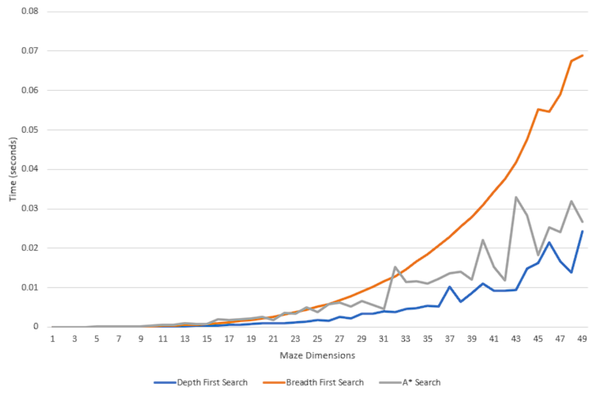
(a) Depth First



(b) Binary Tree

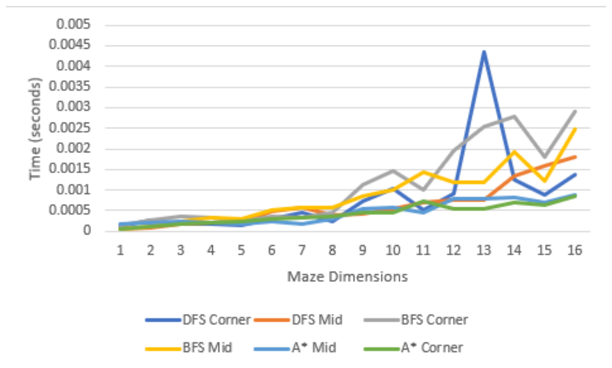


(c) Kruskal's

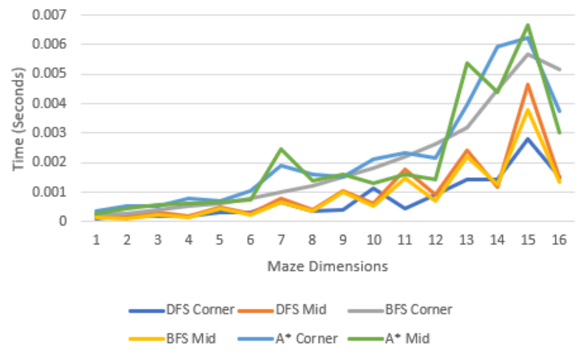


(d) Sidewinder

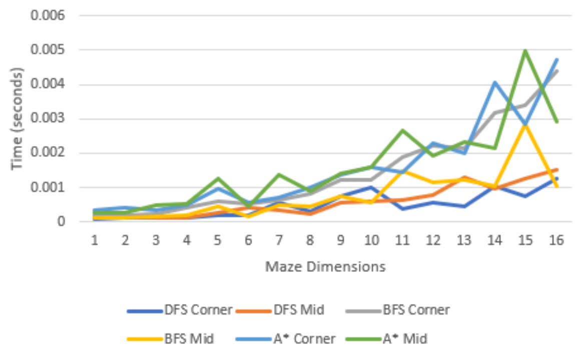
Figure 5: Performance of Pathfinding Algorithms on Various types of Mazes



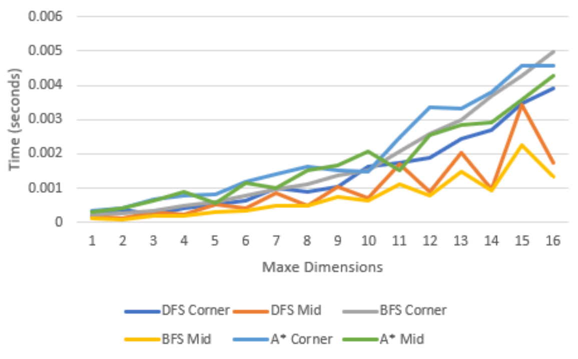
(a) Depth First



(b) Binary Tree



(c) Kruskal's



(d) Sidewinder

Figure 6: Performance of Pathfinding Algorithms on Mazes with differing Goal Locations: Maze dimensions are equal to their labels +10

the maze in this way. This causes it to have to check almost every cell in the maze for a goal node that is in the far corner away from our starting point. Luckily this is not how most human beings choose to solve a maze. If you were to watch someone solve a maze by moving one cell down a particular path before switching to another path and so forth, you might think they had never seen a maze before. While breadth-first search is cost optimal and complete, because we have a perfect maze that only has one solution path, this does not really matter. We know breadth-first search has a time complexity of O(b^{d}) where b is the number of successors from each cell and d is the depth of the solution. The exponential increase in time complexity is very clear in most of the graphs and especially in the sidewinder algorithm. When using breadth-first search, binary tree took the longest time to solve on average, followed by sidewinder, kruskal's and finally depth-first. The other two algorithms, depth-first search and A* search definitely provided more of an accurate read on the amount of time it would take for a human to solve the mazes. Both of the algorithms were fastest on two out of four of the maze generations. With A* being fastest on depth-first and binary tree, and depth-first search being fastest on kruskal's and sidewinder. When the typical person sits down and solves a maze, they follow a single path towards the goal and when they reach an intersection, they first try the path that moves closest to the solution. This is most similar to our version of A* considering the heuristic we chose to use was Manhattan distance which is the distance in "blocks" from the current point to the goal. Therefore it is most appropriate to use a combination of the A* and depth-first search data when determining which mazes would be optimal for our final products. Depth-first search has a time complexity of O(V) where V is the number of nodes in the tree, however the one disadvantage of depth-first search is the memory requirements. Fortunately in our case, we were never solving mazes that were so big that they went out of our memory bounds. A* search has a time complexity that depends on the search heuristic used, but considering that it beats out breadth-first search in every case, we can say that it is less than O(b^{d}). Looking at the time taken by all of our algorithms to solve each of the different types of mazes, depth first was by far the fastest to solve by all of our algorithms, with A* search taking only 5 thousandths of a second for a 49x49 cell maze. Kruskal's and Sidewinder took approximately the same amount of time to solve with both depth-first search and A* search, however, sidewinder took about 1 hundredth of a second longer to compute with breadth-first search at the largest maze size.

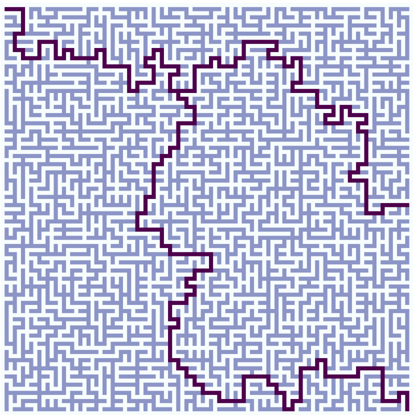
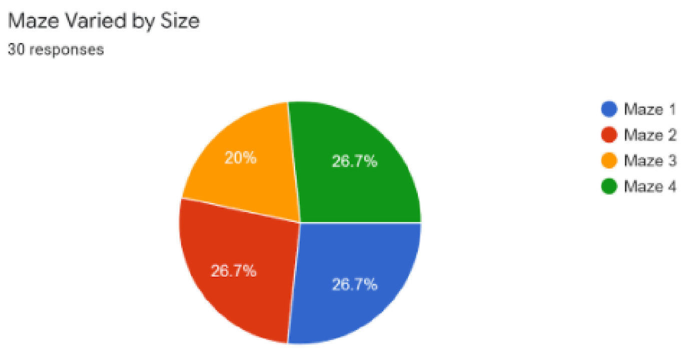


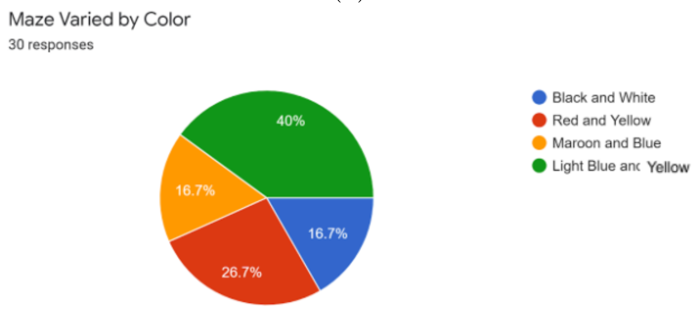
Figure 7: Example of an 50x50 Maze Generated by Kruskals. Visualized with Matplotlib and solved with Depth First search with the solution being in the corner and on the middle of the right side

Looking at Figure 6 we can see the comparisons of the algorithms when run with different goal nodes. Immediately what stands out is the odd zig zag pattern that occurs for depth-first search and breadth-first search when looking for a solution on the middle of the right hand side of the maze. At first I figured that this was an error on our part, however , it does not affect the other algorithms at all which is interesting. In this case, we can surmise that it has something to do with how the algorithms explore the odd vs even rows in the maze. Beyond that, we can notice some similarities between the time taken for the different mazes here and in Figure 5. Depth-first maze generation again had the fastest solve times for all of the algorithms by a large margin (excluding the outlier that occurred at a width of 23). Following this Kruskal's and binary tree both averaged around the same amounts of time taken for each of the runs, while sidewinder took the longest on average for each of the algorithms. This follows a similar pattern to what we noticed in the first test with the sidewinder and binary tree algorithms being high up in the ordering as far as computation time for a solution went. Kruskal's algorithm gave some interesting results when it came to goal nodes being in the middle or corner of the maze. For all of the search algorithms, excluding breadth-first search, the path to the corner goal was found before the path to the goal when it was on the middle of the right hand side. This can not be said about any of the other maze generation algorithms and creates an interesting dynamic when thinking about how to structure our final product. The other aspect of our project analysis was the survey sent out to friends and family regarding the artistic quality of the mazes. We took this data and created charts which can be seen in Figure 7. According to our results, the size of the maze was not a large factor in whether or not people found the maze to be artistically pleasing. The smallest, second smallest, and largest mazes were seen by most people to be the most appealing with the second largest coming in closely behind. As far as color goes, we can see that the majority of people favored the light blue and yellow maze. This maze, which can be seen in the survey, features a lighter pastel color palette compared to the bold colors of the other three. The two colors are also contrasting which would follows from the research we did stating that people often look for contrasting elements in art. In terms of maze type, Kruskal's and depth first had a large majority of the votes over sidewinder and binary tree. This was surprising because sidewinder and binary tree definitely have the most distinctive look, but makes sense considering most people probably have some sort of expectation about what a typical maze should look like.

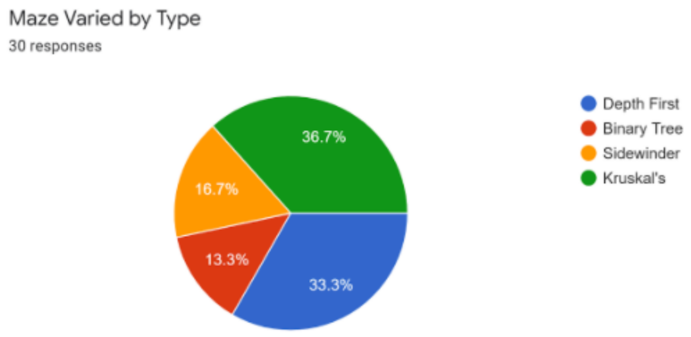
Considering the results from all of our data, it would make sense to primarily generate mazes using Kruskal's algorithm and the sidewinder algorithm for our art. While many people found depth first to be visually appealing, it did not have the complexity that we were looking for in terms of time taken to be solved. Both Kruskal's and sidewinder have unique appearances and scored well when it came to audience appeal, however the main driving factor here was how they performed when it came to solving time. Kruskal's is more versatile considering the variance in solution time depending on the position of the goal node and sidewinder performed well over both test sets.



(a)



(b)



(c)

Figure 8: Results from the Survey Detailing the Preferences of Participants on Artistic Choices for Mazes

7 Conclusion

Assessing all of our results from the different tests we ran, we can determine which algorithms proved most viable for generating a final artistic product. Taking into consideration not only the difficulty and time complexity, but also how they were perceived by the survey participants, we decided to give more weight to the time taken to solve the mazes by our search algorithms over our surveys due to the low amount of survey data that we received. After applying these rankings the two maze generation algorithms that stood out as having the highest potential to be considered an artistic product were Kruskal's and sidewinder. Kruskal's was a favorite among participants in the survey and also performed well in the tests with multiple exits. Sidewinder was less popular aesthetically, but performed better than almost every other maze when it came to the tests that we ran. The vast majority of participants in our surveys did not report that they would be willing to spend money on the product that we currently had, but a substantial amount said that they would if they were improved in some way. Some possible extensions to the project that we could make include adding more maze generation algorithms such as Prim's and recursive division. This would give us more contrast and variety between the different mazes, but still wouldn't give us a truly unique product. In order for us to achieve this, some ideas we had were to create multiple different mazes in the same image where they would randomly take up different portions of the total canvas, or to animate the generation of the maze in a way that allowed for an ever changing image. Before we send any of the images to market, we plan on implementing most, if not all of these additions. Source code and a collection of generated mazes can be seen by going to this Github repository. While we have not yet cracked the code when it comes to computer generated art, we believe that by continuing down this path, we will be able to create art that everyone can appreciate.

List of Figures

References

[1] Nawaf Hazim Barnouti, Sinan Sameer Mahmood Al-Dabbagh, and Mustafa Abdul Sahib Naser. Pathfinding in strategy games and maze solving using a* search algorithm. *Journal of Computer and Communications*, 4(11), September 2016.

[2] Jamis Buck and Jacquelyn Carter. *Mazes for programmers: code your own twisty little passages.* The Pragmatic Bookshelf, 2015.

[3] Christina. Generating and solving mazes with python. *LVNGD*, January 2020.

[4] Cüneyt Dirican. The impacts of robotics, artificial intelligence on business and economics. *Procedia - Social and Behavioral Sciences*, 195:564–573, 2015. World Conference on Technology, Innovation and Entrepreneurship.

[5] Peter Gabrovsek. Analysis of maze generating algorithms. 5, 2019.

[6] Peter Gabrovšek. *Analysis of Maze Generating Algorithms.*

[7] Greg Guibert and Iain Hyde. Analysis: Covid-19's impacts on arts and culture. In *COVID-19 RSFLG Data and Assessment Working Group*, pages 1–3, 2021.

[8] Pavel Hamet and Johanne Tremblay. Artificial intelligence in medicine. *Metabolism*, 69:S36–S40, 2017. Insights Into the Future of Medicine: Technologies, Concepts, and Integration.

[9] Hybesis H.urna. Maze generations : algorithms and visualizations., Nov 2019.

[10] Paul Hyunjin Kim. *Intelligent Maze Generation.* PhD thesis, The Ohio State University, 2019.

[11] Navin Kumar. A review of various maze solving algorithms based on graph theory. March 2019.

[12] Daniele Lizier, Reginaldo Silva-Filho, Juliane Umada, Romualdo Melo, and Afonso Neves. Effects of reflective labyrinth walking assessed using a questionnaire. *Medicines*, 5(4):111, 2018.

[13] Davide Massaro, Federica Savazzi, Cinzia Di Dio, David Freedberg, Vittorio Gallese, Gabriella Gilli, and Antonella Marchetti. When art moves the eyes: A behavioral and eye-tracking study. *PLoS ONE*, 7(5), 2012.

[14] W. H. Matthews. *Mazes and labyrinths: their history and development.* Dover, 1970.

[15] R. Zhou and E. Hansen. Memory-bounded a* graph search. FLAIRS, May 2002.