CS520 - Maze Runner

Alfonso Buono, Jonathan King, Kuber Sethi
ajb393, jmk527, ks1281

1. Project Overview

   Install Instructions:

   1. Go into home directory of folder
   2. Run `pip install --file requirements.txt`
   3. Run `python maze_runner.py`
   4. There will be a doc string including the parameters for each driver

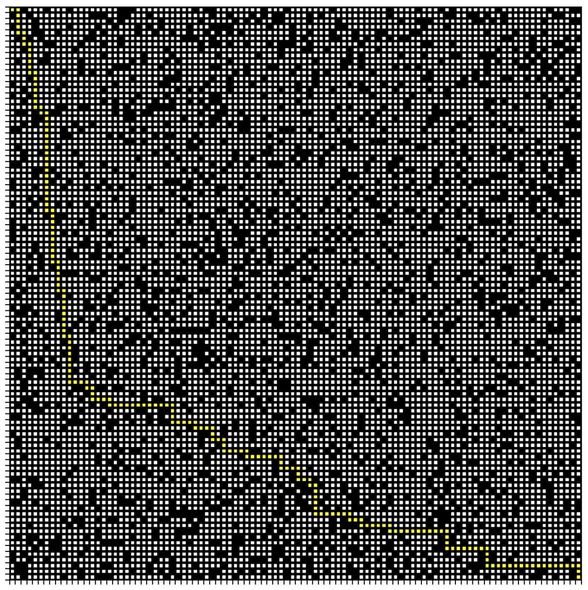
2. Analysis and Comparison

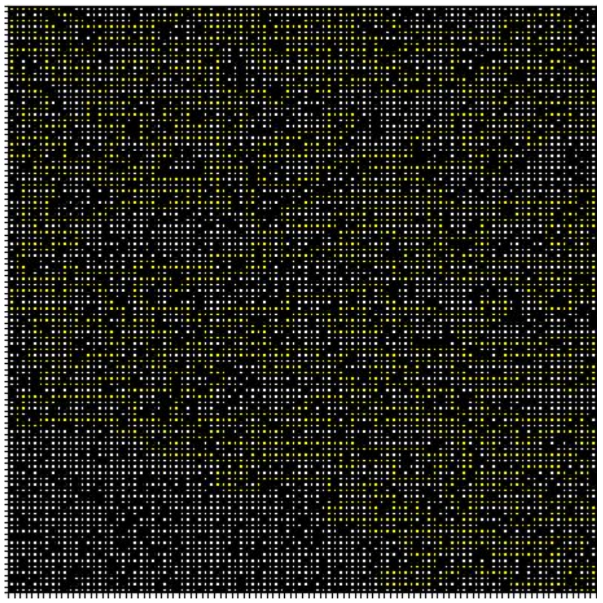
   a. Find a map size (dim) that is large enough to produce maps that require some work to solve, but small enough that you can run each algorithm multiple times for a range of possible p values. How did you pick a dim?

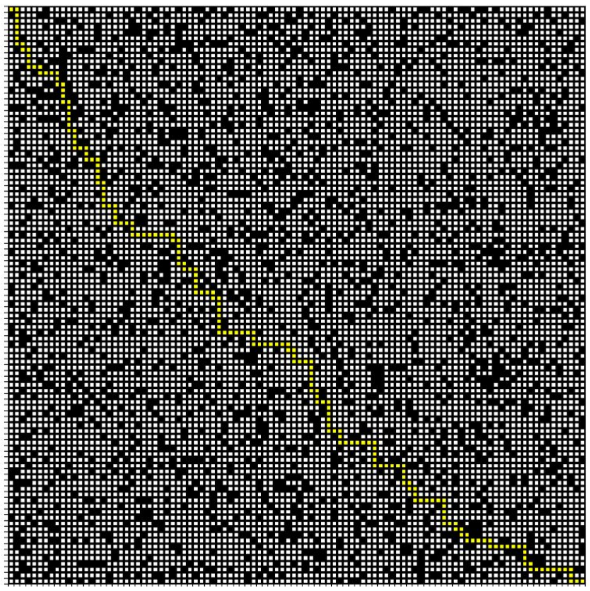
      i. A dim size that worked well for us was 100. It was small enough that we could run most algorithms relatively quickly, and also large enough where we could see the differences in each algorithm. Specifically, we ran each dim size with ten different probabilities, on fifteen mazes. After a certain point, it simply became unfeasible to proceed with this process at higher dim sizes, so we downsized to 100 and it worked fine.

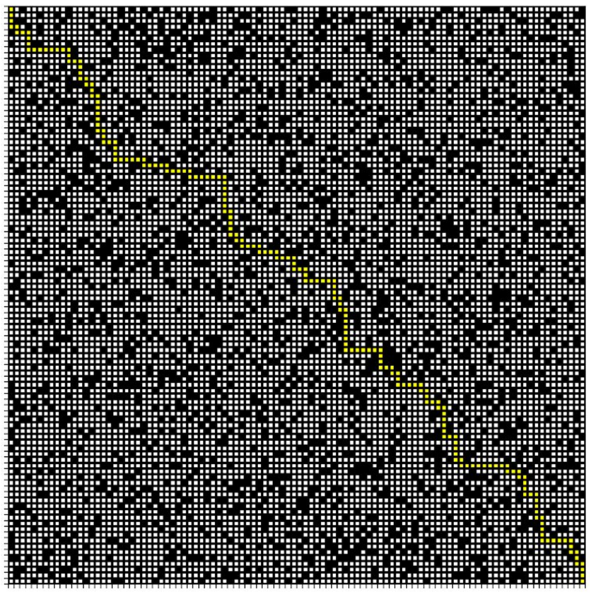
   b. For p = 0.2, generate a solvable map, and show the paths returned for each algorithm. Do the results make sense? ASCII printouts are fine, but good visualizations are a bonus.

      i. The results generally make sense for most of the algorithms, but there are some interesting things we noticed about Breadth-First Search and Depth-First Search.


BFS

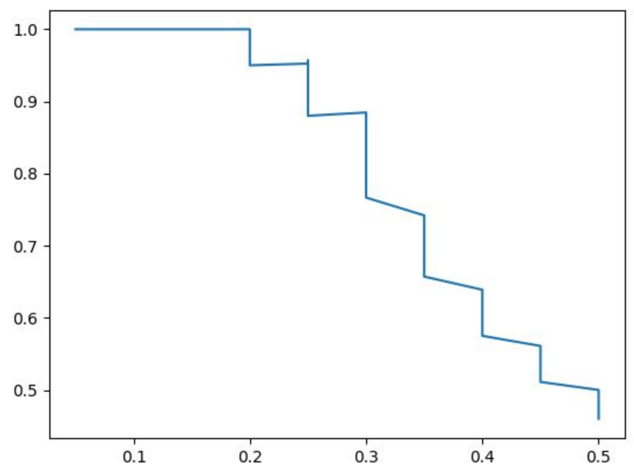

DFS


A* Manhattan


A* Euclidean

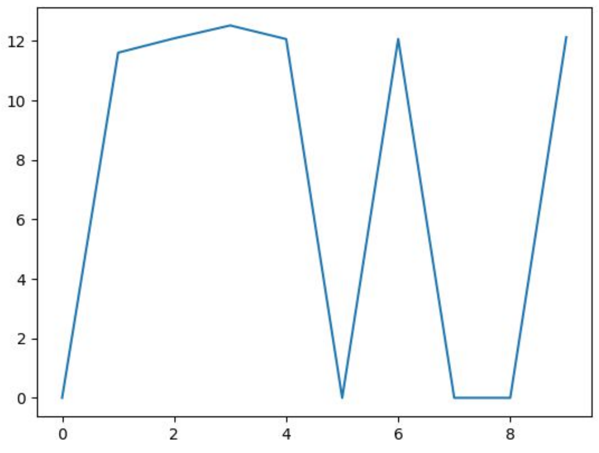
c. Given dim, how does maze-solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability. What is the best algorithm to use here? Plot density vs solvability, and try to identify as accurately as you can the threshold p0 where for p < p0, most mazes are solvable, but p > p0, most mazes are not solvable.
   i. The best algorithm to use here is A* (Manhattan), We found after significant testing that a p value of .3 worked the best when it came to solvability. For p values higher than .3, the solvability rate fell by around 10%, whereas for p values lower than .3, the solvability rate stayed around the same.

d. For p in [0, p0] as above, estimate the average or expected length of the shortest path from start to goal. You may discard unsolvable maps. Plot density vs expected shortest path length. What algorithm is most useful here?
   i. The algorithm that was most useful in this case was A* (Manhattan), as it guarantees the shortest path, and is also reasonably fast due to the heuristic using the Manhattan method. Additionally, for p values between 0 and .3, it performs well as it is able to move around the walls in an efficient manner.

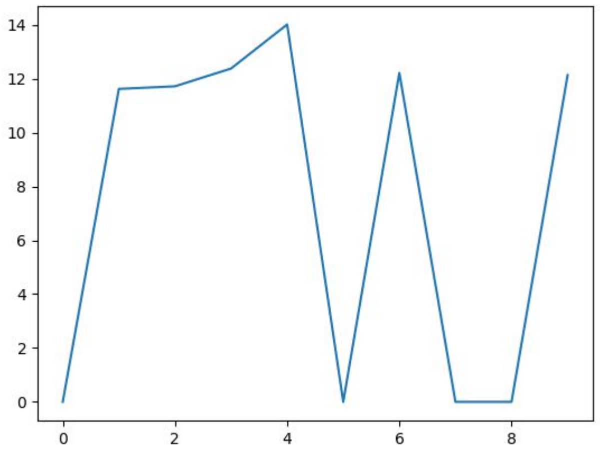


Density vs Solvability (x = p, y= % maps solved)

e. Is one heuristic uniformly better than the other for running A*? How can they be compared? Plot the relevant data and justify your conclusions.
    i. The Manhattan heuristic is generally better at solving this problem, for quite an intuitive reason -- the search algorithms have to move in a 'block by block' manner. However, our data reflected that in small map sizes, the runtime difference was negligible. At higher dim sizes (dim > 100), the Manhattan heuristic performs noticeably better, and we believe that this trend would continue for increasing dim sizes. On average the Manhattan heuristic performed roughly 5% better than the Euclidean heuristic, but the improvements were not uniform.



Manhattan (x = iteration with increasing dim size, y = seconds)



Euclidean (x = iteration number with increasing dim size, y = seconds)

f. Do these algorithms behave as they should?
    i. The algorithms all behave correctly, but we noticed that performance wasn't as expected in some cases. For example, even though the Big O of BFS and DFS is the same, DFS performs noticeably faster. This is likely due to the specific implementation and data structures used, such as the Deque queue structure in BFS. Theoretically A* Manhattan should be faster than A* Euclidean, due to the nature of the problem, as the maze is essentially traveled 'block by block', but our data shows that they are very similar. In addition to this, the implementation of A* did not utilize the set as it used a "Node" object to keep track of the path. Therefore the visited lookup was O(n) as opposed to O(1). This was a design decision to make keeping track of the path much simpler.

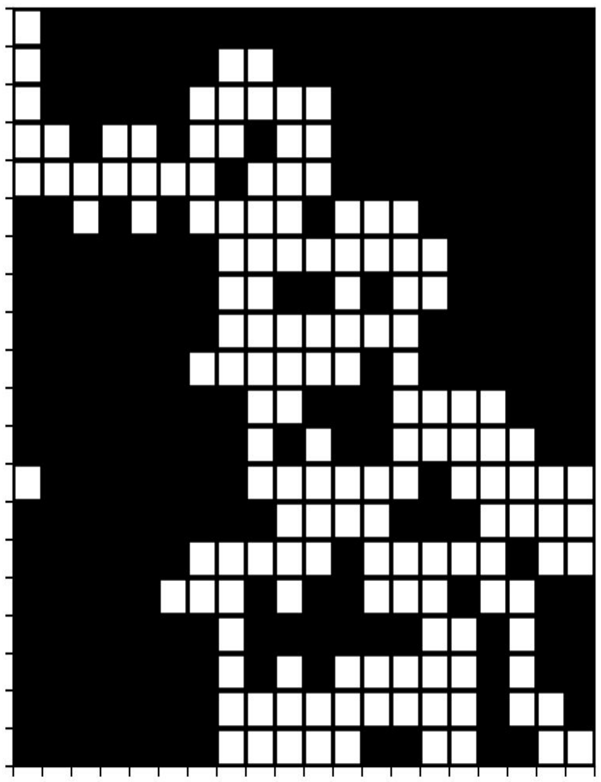
g. For DFS, can you improve the performance of the algorithm by choosing what order to load the neighboring rooms into the fringe? What neighbors are 'worth' looking at before others? Be thorough and justify yourself.
    i. Yes, in DFS you should always choose to load the neighbors to the bottom and to the right, only afterwards loading the neighbors to the top and left. This will cause the bottom and right nodes to be popped first and therefore they will be expanded first. This should be prioritized because these nodes are closer to the goal point. In a way, this is similar to how A* Manhattan prioritizes searching nodes that are closest to the goal. In most cases, simply going down and right is the fastest way to reach the goal, unless the wall probability is extremely high, so it makes the most sense to prioritize these nodes the most.

h. Bonus: How does the threshold probability p0 depend on dim? Be as precise as you can.
    i. We believe that the p0 threshold does not depend on dim at all. Regardless of the value of dim, p0 will yield the same density in the maze.

3. Generating Hard Mazes
    a. What local search algorithm did you pick, and why? How are you representing the maze/environment to be able to utilize this search algorithm? What design choices did you have to make to make to apply this search algorithm to this problem?
        i. We chose to use the Simulated Annealing algorithm to generate hard mazes. We did this because we felt that it made the most sense in order to find a local maximum of our difficulty function. Other algorithms, such as the Genetic Algorithm, had a higher chance of generating mazes that were unsolvable. In addition, we believed that it made too large of jumps around the fitness function. With Simulated Annealing, we were able to move between different maze states in smaller intervals. In our algorithm, we defined a neighboring state to be a copy of the current maze with now another coordinate set to be a wall. This way, we were able to change our mazes one coordinate at a time.

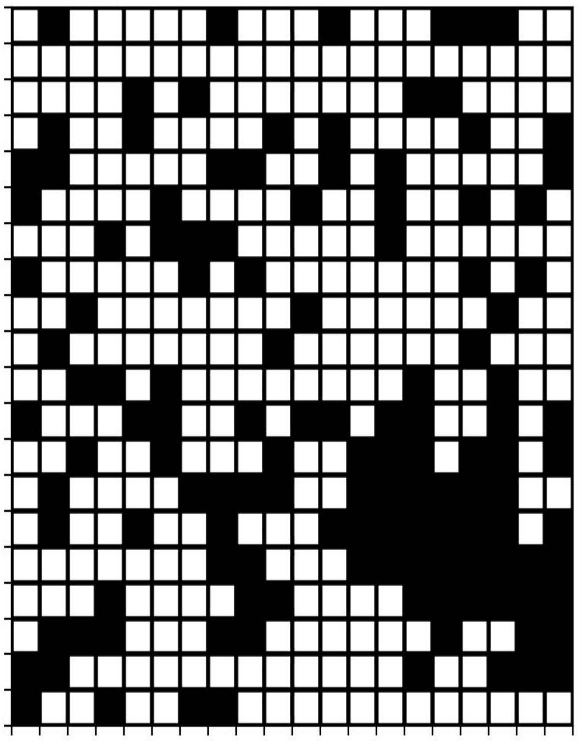
    b. Unlike the problem of solving the maze, for which the 'goal' is well-defined, it is difficult to know if you have constructed the 'hardest' maze. What kind of termination conditions can you apply here to generate hard if not the hardest maze? What kind of shortcomings or advantages do you anticipate from your approach?
        i. Our termination condition was defined to be when a maze no longer had any valid neighbor states. We used a set of all possible new wall locations to keep track of this. For example, when generating a neighboring state, we would pop a new wall location from this list and set it in the maze. If this new maze was unsolvable, we would revert to the previous maze, pop a new wall location, and set that location to be a wall. We would repeat this process until we popped a wall location that still yielded a valid maze. The advantage of this approach is that it was much quicker. As we popped possible wall locations out of our set, we decreased the number of wall locations that we would check later in the algorithm. However, there were some shortcomings to this. For example, there is no way of removing walls and trying new ones. Once a wall has been placed, a neighboring state has to have that wall as well. This decreases the amount of possible maze states.

c. Try to find the hardest mazes for the following algorithms using the paired metric: DFS with Maximal Fringe Size & A*-Manhattan with Maximal Nodes Expanded

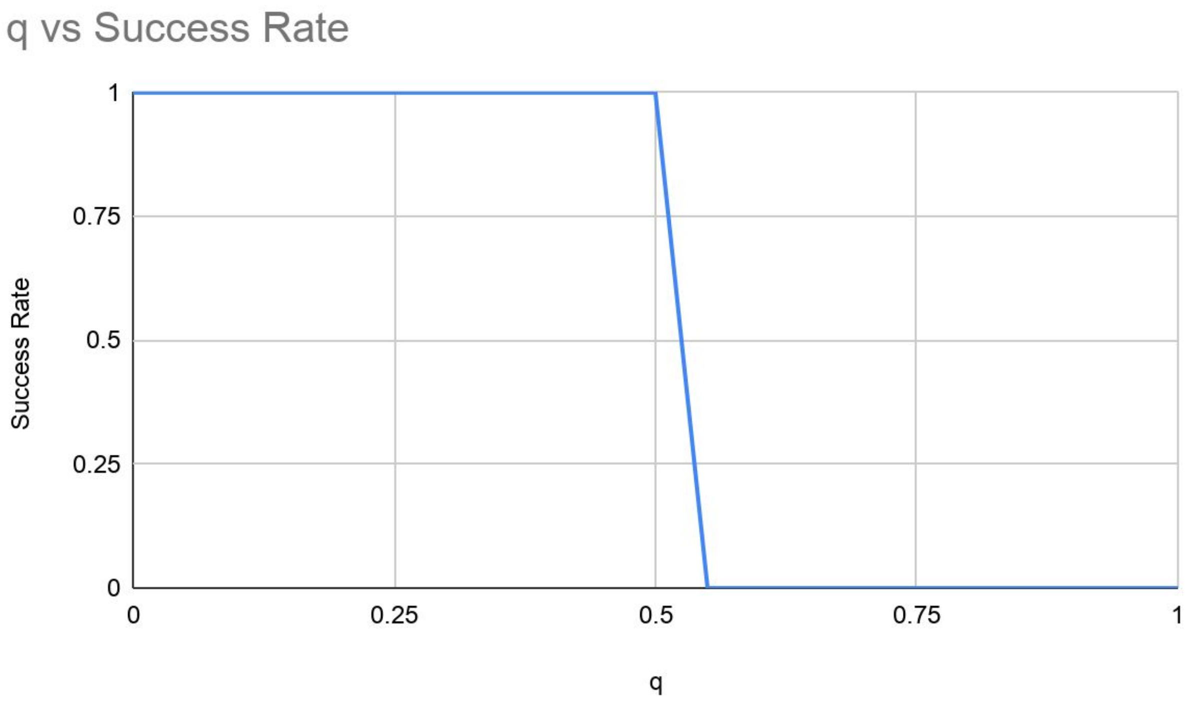


The graph to the left is a maze that was generated with:
dim = 20,
initial wall probability = .2,
search = dfs,
difficulty metric = max fringe size.
This graph makes sense because, as you can see, there are many loops. In addition, there are also more open regions in areas that are far away from the start point. DFS will naturally explore areas that are farthest from the starting point, but as you can see from the graph, there is only one opening path to the region that leads to the goal. DFS will conduct many searches around the open regions before finding this opening, hence creating a large fringe size.

The graph to the right is a maze that was generated with:
dim = 20,
initial wall probability = .2,
search = a_star_manhattan,
difficulty metric = max nodes expanded.
This graph makes sense because the only path leading to the goal goes through (0,9). This node is in the middle left region of the graph. Due to our heuristic, A* Manhattan will tend towards nodes in the bottom right section of the graph as this is closer to our goal. In this maze, A* manhattan is likely to get stuck in the open region to the right of the graph before it discovers the entranceway on the left. There is also a dead end in the bottom right of the graph on the opposite side of this entranceway.

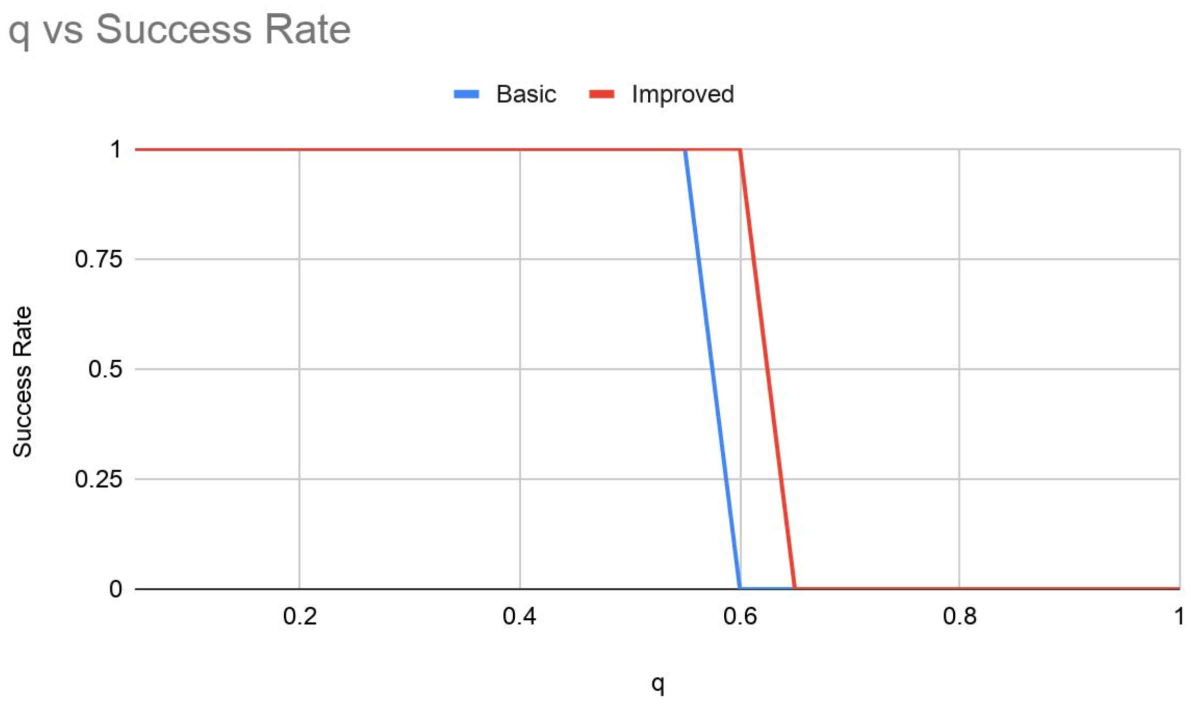


4. What If The Maze Were On Fire?
    a. How can you solve the problem (to move from upper left to lower right, as before) in this situation?
        i. “As a baseline, consider the following strategy: simply find the shortest path in the initial maze, and run it; if you die you die.”
    b. At density p0 as in Section 2, generating mazes with paths from upper left to lower right and upper right to lower left, simulate this strategy and plot, as a function of q, the average success rate of this strategy.



        i. Tested with dim size of 100 and p0 = .3.
    c. Can you do better? How can you formulate this problem in an approachable way?
        i. The given “algorithm” of just following the premade path no matter if facing death or not can be improved. The idea that we came up with is by recalling our search algorithm. This would occur whenever the runner’s next move is into fire. This is because the runner cannot go through fire and thus the path has been cut off. When recalling the algorithm it attempts to find another path avoiding the fire, by treating them as walls.
    d. How can you apply the algorithms discussed?
        i. We can apply the algorithm discussed by checking the next cell that is going to be moved into. If there is a fire there, then instead of dying, like in the basic algorithm, we are going to rerun the search algorithm. If there is a path to the end of the maze then we move to the first cell in that path. If there is not, then we return as there is no end to the maze (since the fire cut out all the possible routes to the end of maze)

e. Build a solution, and compare its average success rate to the above baseline strategy as a function of q.



   i. The comparison between the two shows that our small change increases the success rate but not by very much. This is because although it reruns the algorithm, either the fire cuts off all of the possible paths or we move into a cell that is next to fire. Even though that cell is “safe” since it is not on fire, it is dangerous since fire can immediately spread to that cell next iteration.

f. Do you think there are better strategies than yours? What would it take to do better?

   i. Since we saw that our new algorithm does not improve the success rate that much we decided that it can further be improved by changing the heuristic when using A*. This is because we can make it so that blocks that have neighbors with fire have a lower heuristic value and thus the algorithm would hopefully ignore those cells. Doing so will let the runner solve the maze but also have it be more inclined to not go to cells that would potentially put it into a dangerous situation. Going into cells by fire increases the risk of catching fire and dying.