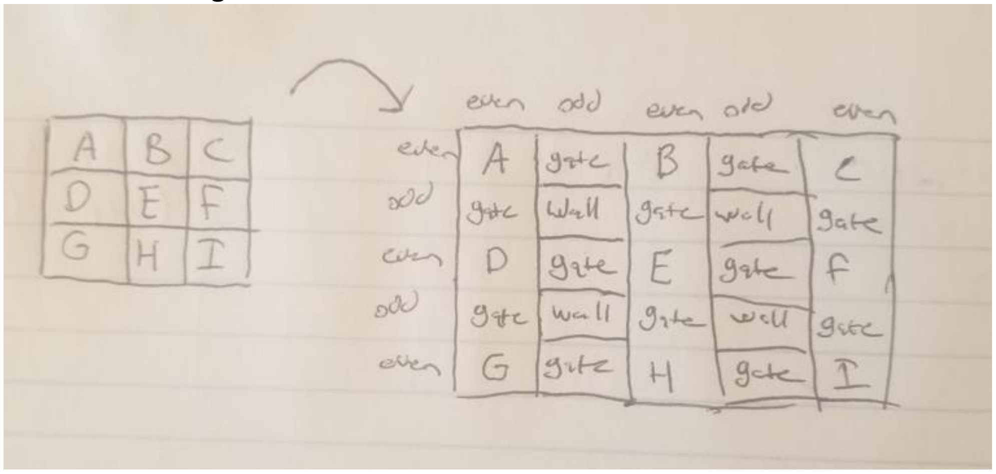
Sam Beals
CIS 365
Generating Mazes and Finding Solutions

Kruskal's Algorithm

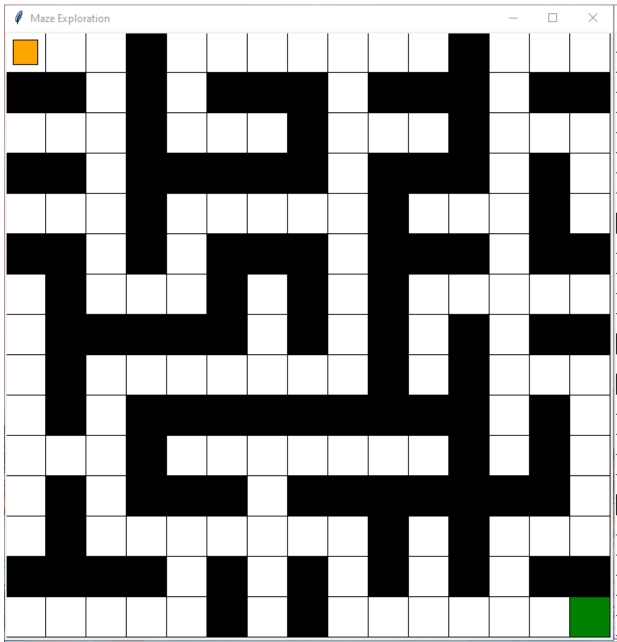
After looking into several algorithms, I ultimately ended up implementing Kruskal's algorithm for my maze generation needs. In a nutshell, Kruskal's algorithm randomly looks at walls between cells and determines if that particular wall would create a redundant connection between cells that are already connected, in which case the wall remains, or if it would not create a redundant connection it removes the wall creating a path to a new cell (or group of cells). In this way a maze is created that has a path from each cell to each other cell.

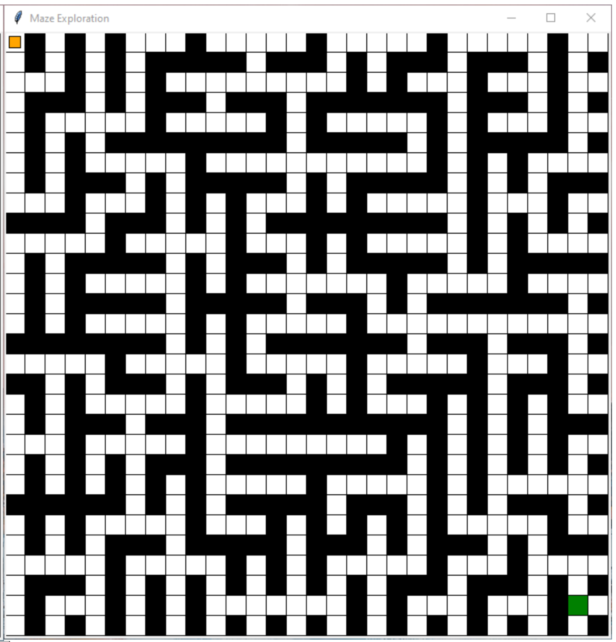
The thing that drew me to Kruskal's algorithm over others was that as I read the steps, I was able to quickly form a mental model for how iterating through the walls and checking if the cells are already connected or not would translate to code. Basically, there is an array of "groups" that contains each "group" which is also an array containing the coordinates of each cell in that group. As you loop through the walls you check if the cells connected by that wall are already in the same group, if they are you leave it as a wall and move on. If they are not and are each already connected to different groups you remove the wall and connect the two groups, and if neither are in a group you remove the wall and create a new group with the two cells being connected. By the end you have just one big connected group of all cells.

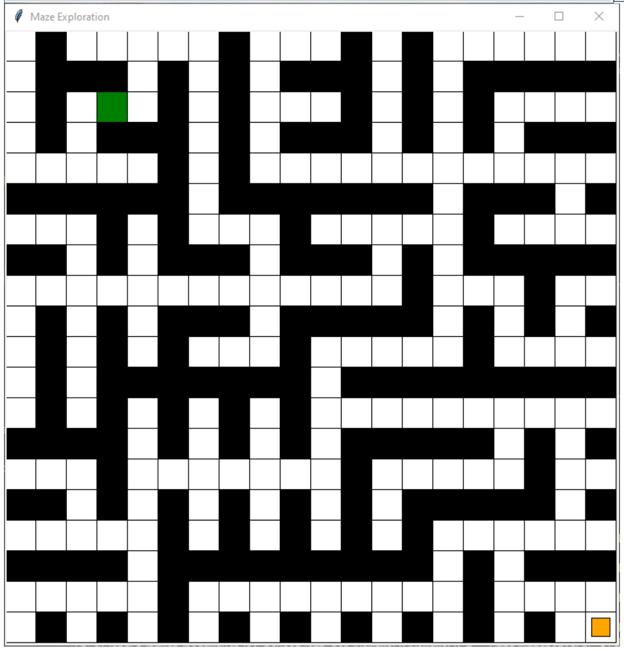
The main challenge in implementing this algorithm was figuring out how to take the theoretical concept of the algorithm, where walls are infinitely thin, and apply it to our Tkinter "world" where walls exist as full cells between other cells. In order to achieve this, I eventually came up with the following translation:



As I looped through and create the cell in each row and column, I noticed that if the indices of both the row and column are even I can create an open cell, if both are odd I always create a wall, and if one is even and the other odd or vice versa I add it to a list of "gates". These "gates" may potentially be open cells or closed walls, and so these are what I iterate through as I apply Kruskal's algorithm in order to create a functioning maze. Here are some examples of the mazes this can generate:


Maze Exploration


Maze Exploration


Maze Exploration

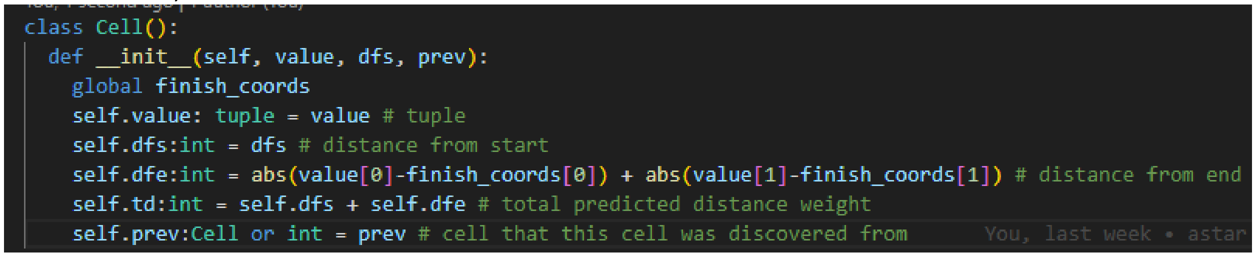
A* Algorithm

For the first maze solving solution I went with the A* algorithm. This algorithm keeps track of data for each cell as it is "discovered" from a neighboring cell including how far was traveled from the start to reach it and an estimated distance from the goal based on some heuristic. For this heuristic I went with the "Manhattan distance" from the goal cell, which is the sum of the total distance of width and height from the current cell to the goal cell. As cells are discovered they are added to an "open queue", and as you move into a cell and discover its neighboring cells you move the current cell from the open queue to the "closed queue". As you progress you continually check the open queue to see which spot has the lowest predicted total distance by adding together the distance from the start to the predicted distance to the end. This way you are constantly moving from the cell that your heuristic tells you has the best chance of leading you on the shortest path to the finish line.

I liked the idea of the A* algorithm because it seemed to be a simple but effective implementation of applying estimations in order to guide the search in a way that is often more efficient and gives the appearance of a sort of "intelligence" behind the decision-making pattern. I implemented a version of the A* algorithm where you continue checking the neighbors of the cell you are currently on until you reach a point where there are no new neighbors to check (a dead end), at which point you refer back to the open queue and find the cell in the queue with the minimum predicted total distance.

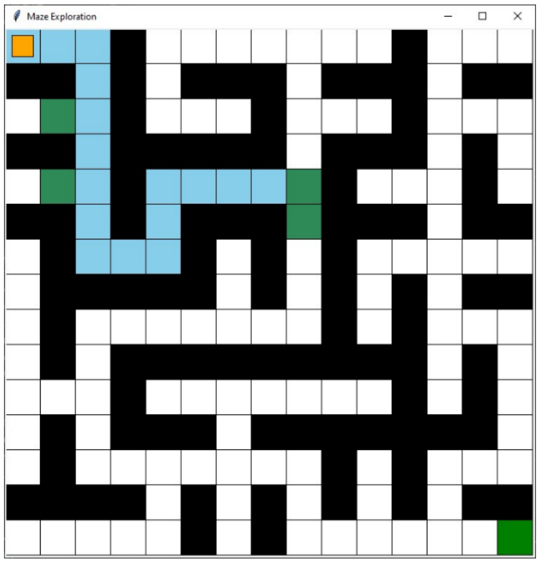
The key difference between the A* algorithm and Tremaux's algorithm is the heuristic used by the A* algorithm. For both algorithms you keep track of data on each cell as it is traversed, but in A* there is a more straightforward calculation going on resulting in predictive values, whereas in Tremaux we are just "marking" cells and keeping track of if they have multiple paths out or not.

In order to track the data needed for the A* algorithm for each cell I created a "Cell" class that kept track of the value of the cell, it's distance from the start, it's predicted distance from the end, and also which cell this cell was discovered from.

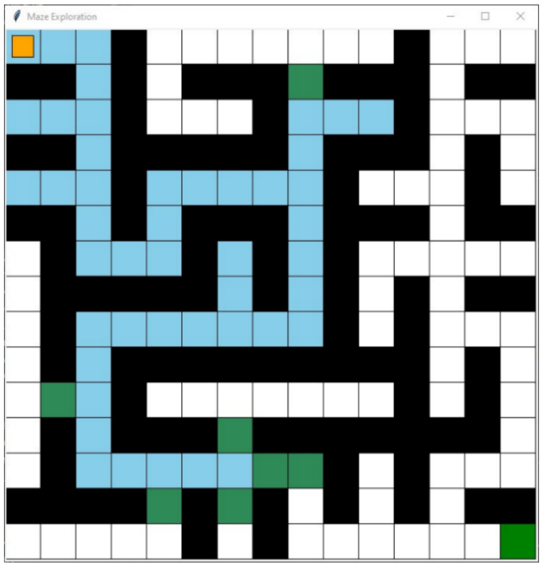


```python
class Cell():
  def __init__(self, value, dfs, prev):
    global finish_coords
    self.value: tuple = value # tuple
    self.dfs:int = dfs # distance from start
    self.dfe:int = abs(value[0]-finish_coords[0]) + abs(value[1]-finish_coords[1]) # distance from end
    self.td:int = self.dfs + self.dfe # total predicted distance weight
    self.prev:Cell or int = prev # cell that this cell was discovered from      You, last week • astar
```

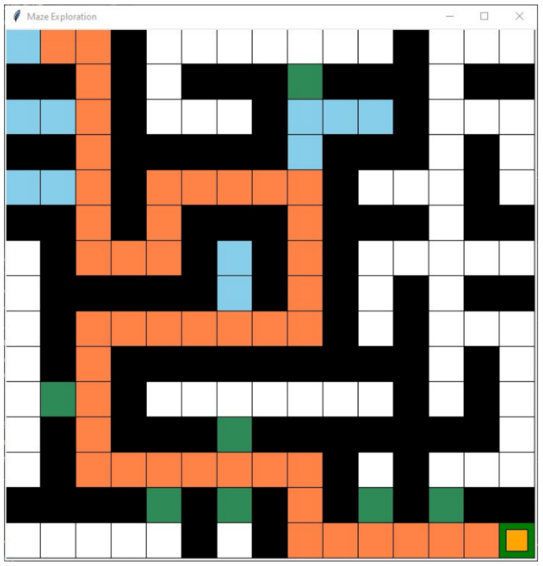
By keeping track of the previous cell, we are able to traverse backward from the end once we find it in a similar way to how a linked list would work, tracing the path back from the end to the start and building out the path as we go. I was able to set up the open and closed queues as arrays of these Cell objects and pulling the object with the minimum total predicted distance from the open queue to continue the search. The results are as follows:



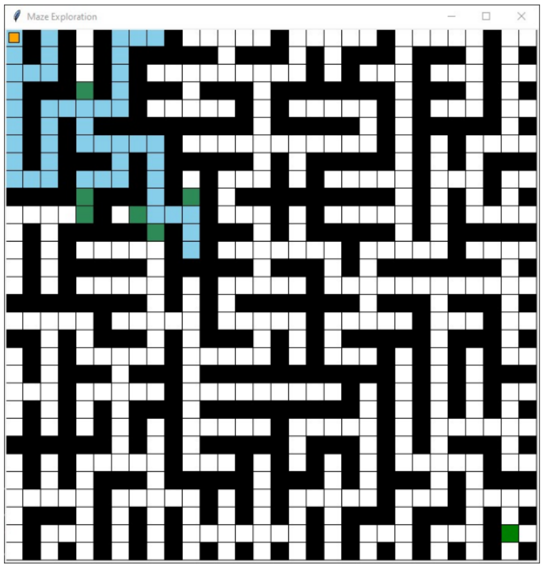
As we step through, we keep track of the closed queue (blue) and open queue (green)



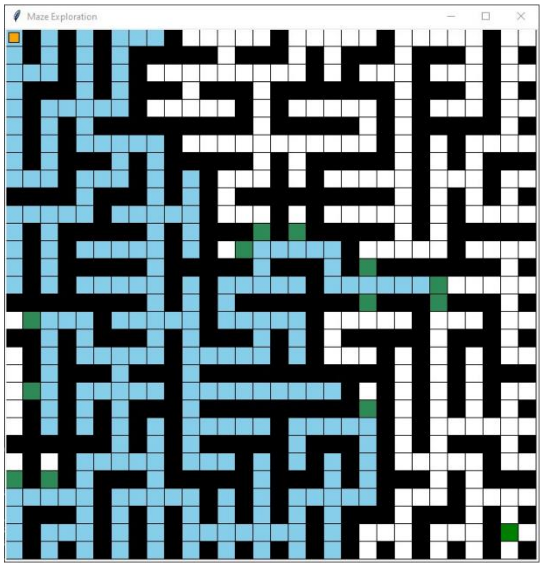
As long as our path is relatively straight the heuristic keeps things moving in the right direction



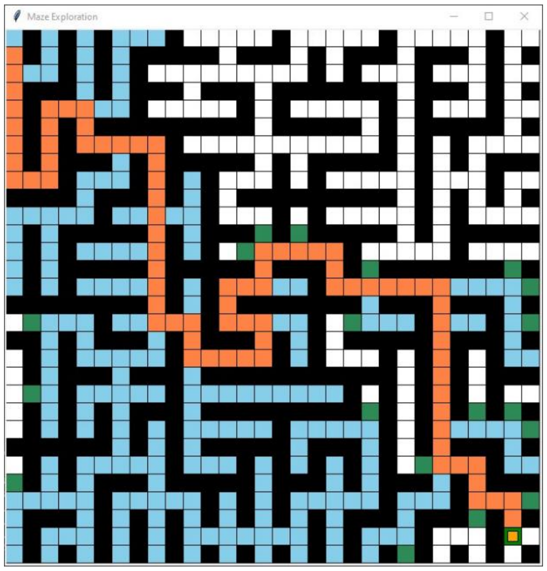
Eventually we reach the end and walk back to the start, giving us the final path



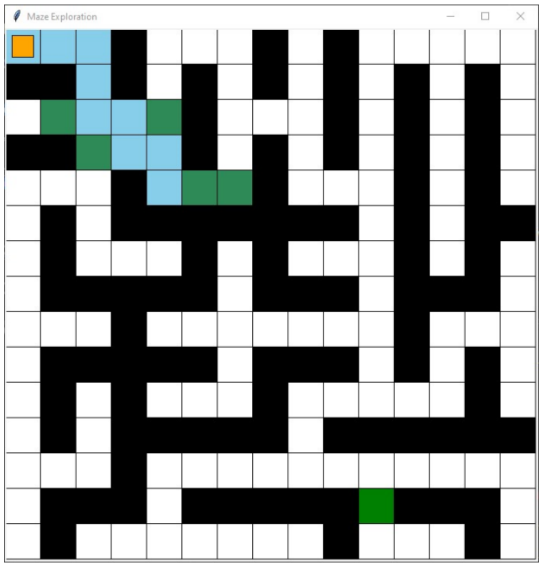
On a more complicated path there are more chances for the algorithm to go back and try previously undiscovered paths



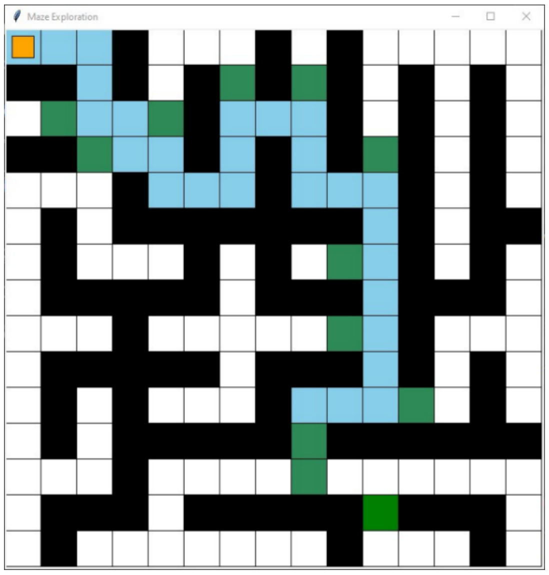
As the path deviates from a straight line and involves moving away from the goal, more paths off to the sides get explored



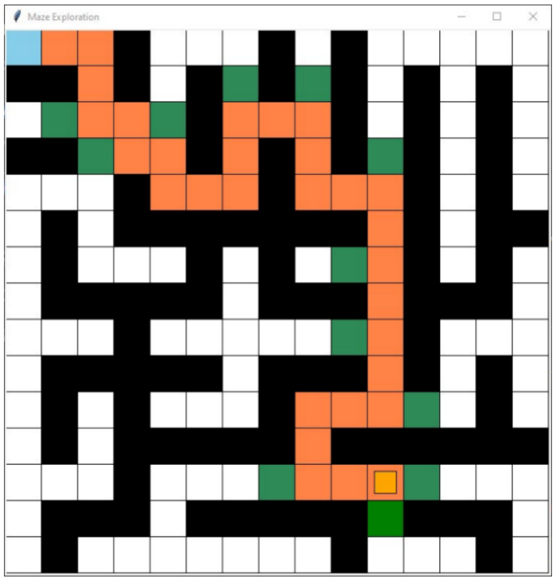
Our heuristic keeps us from going off on unnecessary tangents as we zero in on the end



The algorithm is not troubled by open sections, it is able to continue stepping forward regardless



No matter how much open space it would be able to keep track of which cell each cell is discovered from, therefore no changes are needed to get A* working with paths that have multiple entrances or wide areas.



An efficient path is still able to be found

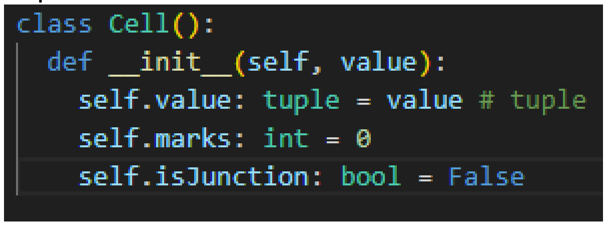
Tremaux's Algorithm

Unlike the A* algorithm, Tremaux's algorithm has no heuristic and does not require the ability to keep track of which previously discovered path has the lowest predicted distance and teleporting back to those spots. As such, it more closely emulates how a human being with limited information would be able to find their way through a maze, assuming they had a way to "mark" cells as they passed through. The basis of the algorithm is that as you pass through each cell you "mark" it, and these markings inform decisions you make when you reach points that have multiple paths that could be taken out of them. I have taken to referring to these cells as multiple possible paths out as "junctions", and the pattern for what to do once you reach a junction is as follows:

1. If there are any paths out of the cell that have no markings pick one at random and travel down that path until you hit another junction or the end.
2. If there are no paths that have no markings, pick one that has one mark on it and go down that one, marking each cell in the path a second time.

As you step through the algorithm paths that lead to dead ends end up with two marks and can from then on be essentially ignored, they are ruled out for being included in the path to the goal. Choosing unexplored paths randomly, you will eventually find the goal and will have left behind you a path of cells marked with one marking that can be traced back to the start, providing the ultimate path.

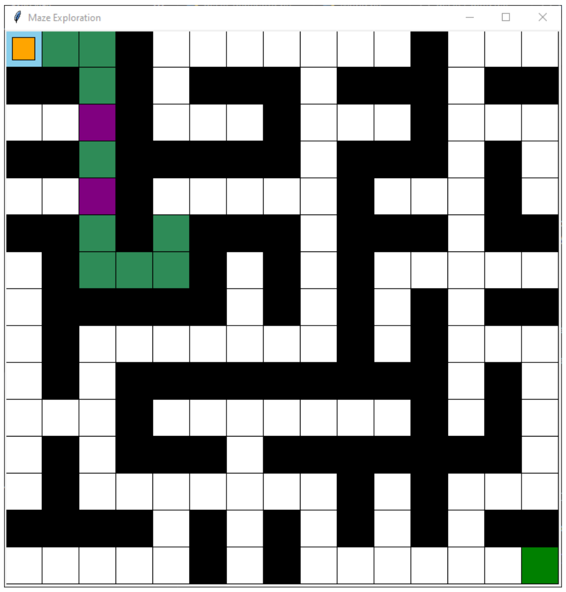
The main challenge in implementing Tremaux's algorithm was in finding "junctions" and determining the hierarchy of decisions to be made at each, as well as keeping track of the path from beginning to end throughout the process. Like my solution for A* I utilized a "Cell" class to help determine how each cell should function:



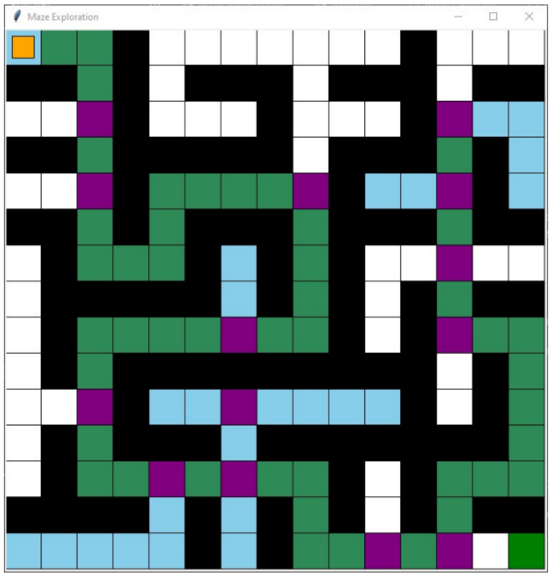
```python
class Cell():
    def __init__(self, value):
        self.value: tuple = value # tuple
        self.marks: int = 0
        self.isJunction: bool = False
```

Much simpler than the A* Cell, the Tremaux Cell is there to keep track of which cells are junctions and therefore decision making points, as well as keeping track of the marks on each discovered cell.

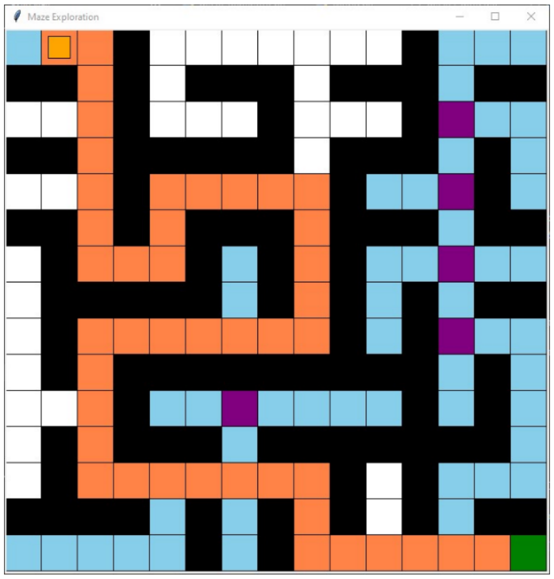
Because you don't know if the path you are arriving from will eventually be ruled out, it does not make sense to mark cells as having a previous cell in the same way the A* cells did. Instead, I find the path by making two separate trips from the start to the end; one to go through and mark cells until the end is found, and a second trip to follow the cells with one marking, recording each in an array for the path until arriving at the end. As you step from cell to cell, if a junction is found you step to that junction and to the cell off of that junction that is not yet in the path. However, this causes some problems when we modify a maze to have open spaces instead of the path being only one cell wide. Open spaces create blobs of junctions, and so I had to modify the code to get this to work. I ended up coding it so that if you enter a junction with no new cell with 1 marking on it, but instead other junctions in a "blob", you step randomly around from junction to junction until the path out is found. This is not a perfect solution, but eventually manages to find the way out and continue to find the path.



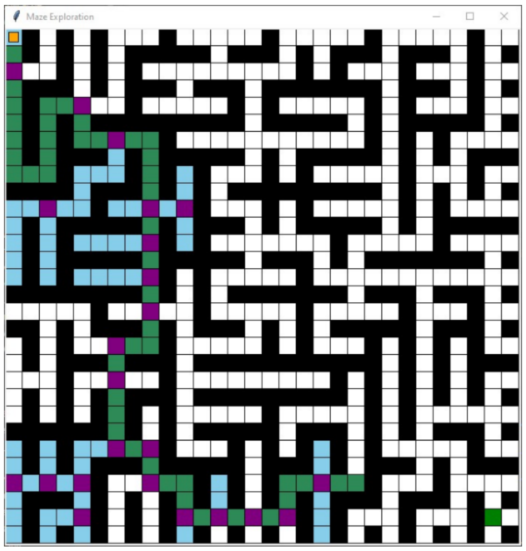
As we step through, we mark cells once (green) and mark junctions (purple)



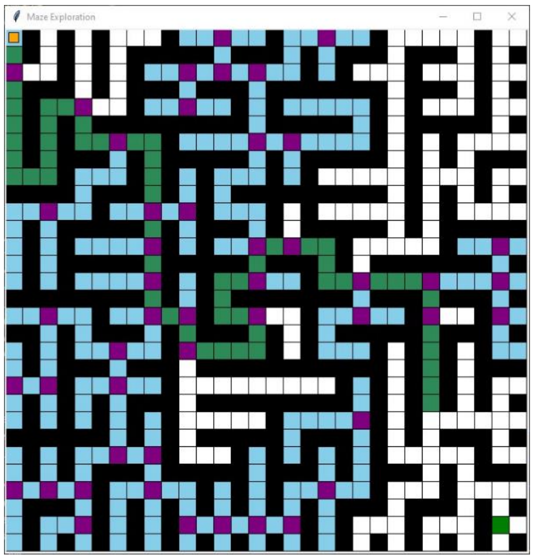
As we return from paths that lead to dead ends, they are marked a second time (blue) and algorithmically are ruled out and not traveled down again



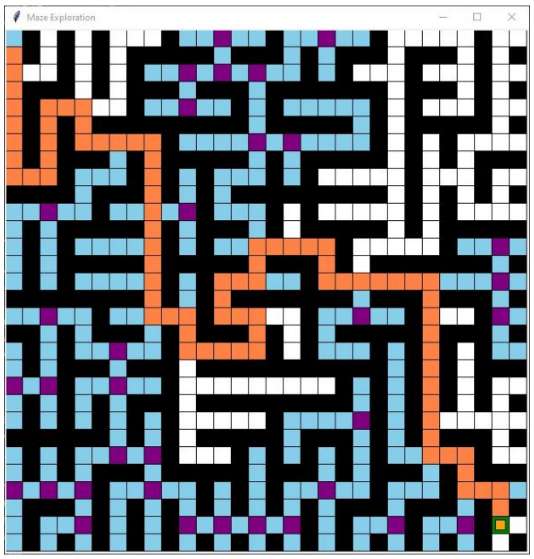
Paths are chosen randomly so there may be some unnecessary distance traveled before reaching the end.



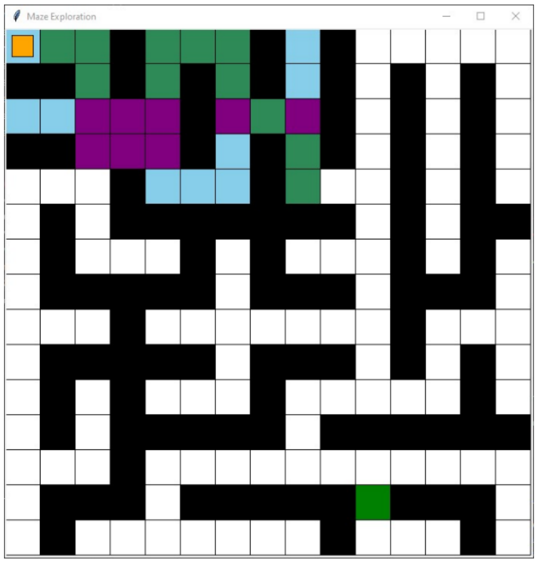
This applies on a larger maze as well



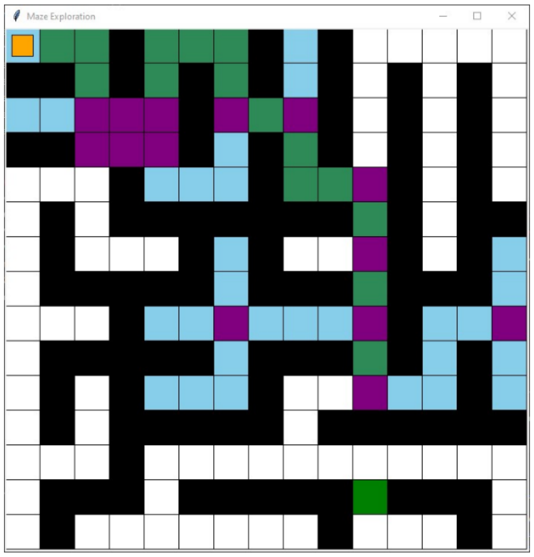
Eventually incorrect paths are pruned off and we are left with a single path one mono-marked cells



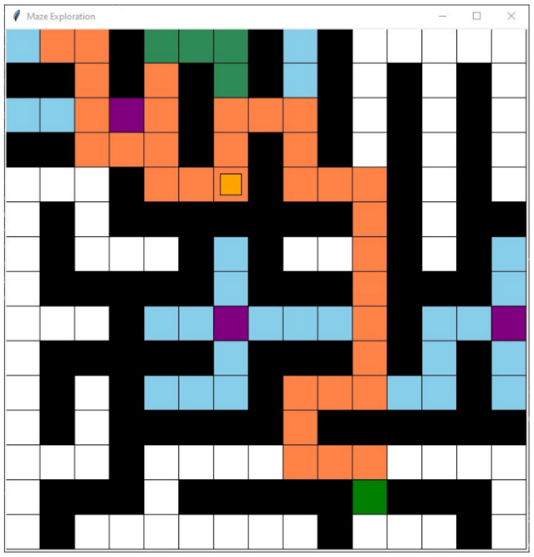
Eventually we can trace back the path from beginning to end to find the solution



Open spaces cause junction blobs that are difficult to trace through after our initial pass through to mark a path



There is still a viable path through



We are left with a less efficient path, but one that gets us to the end eventually