Assignment 1: Fire Maze Report

Karun Kanda (Section 7), Agam Modasiya (Section 3), and Varun Gaddam (Section 3)

Due: Feb. 19th 2021

1 Usage

To run DFS, BFS or A* on a static maze in the terminal type in:

```
python MazeGUI.py <dimension size> <probability of generating an obstacle> <algorithm>
```

To run strategy 1, 2, or 3 on a dynamic maze, type in the terminal:

```
python MazeGUI.py <dimension size> <probability of generating an obstacle> <strategy number> <
    probability of generating fire>
```

2 Language and Libraries Used

For this project we decided to go with Python and we used the following Python Libraries:

1. pygame - to create a visual for the maze solving strategies.
2. sys - to obtain command line arguments so we can dynamically create the mazes based on a dimension size and probability size.
3. random - used when generate the fire and to generate the obstacles when creating the static mazes or the dynamic mazes.
4. numpy - used to create 2D arrays to represent various states of the mazes.
5. math - to calculate the probability of putting a obstacle in the maze and to calculate the probability of generating the fire for the fire maze.
6. time - used to let the system rest when generating the dynamic mazes and to time the search algorithms and strategies.
7. collections - used to queue library from this library to implement Breadth First Search on.

3 Questions about Project 1: (Analysis)

3.1 Problem 1: Write an algorithm for generating a maze with a given dimension and obstacle density p.

To generate the maze with a given dimension and obstacle density p we had to first figure out how many obstacles were to be generate in the maze. The amount of obstacles would have to fill a certain percentage of the maze based on the area of the maze. For example, if the maze had an area of a 100 (with dimension size 10) and a probability of 0.1 then only 10 spots ($100 * 0.1 = 10$) in the maze would have to be filled. To represent the maze we choose to create a 2D matrix where the ith row and jth column would represent a spot in the maze. First we will create a variable **obstacles** $= (dim * dim) * probability$. Then iterate this until it reaches zero where in each iteration we will choose a random index $|i \wedge j| \in$ dimension and create an array that contains 0 or 1. Then based on the condition randomly choosing between 0 and 1 we get 0, obstacles we need to generate isn't 0 and the array representing the maze contains a $0 \rightarrow$ occupy the cell with a 1 to show there is an obstacle and decrement the number of obstacles place.

Code:

```
# iterate based on the amount of obstacles that are left, when there are no obstacles left then
    draw the maze
while obstacles != 0:
    i = random.choice(dim_array)
    j = random.choice(dim_array)
    if i == 0 and j == 0: # this is what we will define as a start node with yellow
        pass # cant place an obstacle in the starting
    elif i == size - 1 and j == size - 1:
        pass # can't place an obstacle in the ending
    else:
        arr = [0, 1] # these will represent random choices
        if random.choice(arr) == 0 and obstacles != 0 and tracking_array[i][j] == 0:
            tracking_array[i][j] = 1
            obstacles -= 1
```

Next step is to draw the maze visual using pygame. The basic idea of drawing a maze in pygame is we need a x, y, cell size to draw one rectangle then to draw a full maze it would consist of drawing rectangles for a certain maze size that we were given an argument too (dimension size). Then it would be simply iterate in a nested loop for a certain amount of rows and columns to draw dimension X dimension maze. The start of the maze $i = 0$ and $j = 0$ we want to draw a yellow rectangle, the end of the maze $i = size - 1$ and $j = size - 1$ we want to draw a green rectangle. If the maze representation at index i and j equals 1 then draw a full black rectangle. Otherwise draw the outline of the maze. Then at the end we would update the display.

Code:

```
# draw the visual
for k in range(0, size):
    self.x = 5
    self.y += 5
    for b in range(0, size):
        if k == 0 and b == 0: # this is what we will define as a start node with yellow
            cell = pygame.Rect(
                self.x, self.y, self.cell_size, self.cell_size)
            pygame.draw.rect(screen, YELLOW, cell)
        elif k == size-1 and b == size-1: # this will define the ending node (G)
            cell = pygame.Rect(
                self.x, self.y, self.cell_size, self.cell_size)
            pygame.draw.rect(screen, GREEN, cell)
```

```
        elif tracking_array[k][b] == 1: # this will define an obstacle
            cell = pygame.Rect(
                self.x, self.y, self.cell_size, self.cell_size)
            pygame.draw.rect(screen, BLACK, cell)
        else: # anything else draw the maze design
            cell = pygame.Rect(
                self.x, self.y, self.cell_size, self.cell_size)
                pygame.draw.rect(screen, BLACK, cell, 1)

        self.x += 5
pygame.display.update()
```

3.2 Write a DFS algorithm that takes a maze and two locations within it, and determines whether one is reachable from the other. Why is DFS a better choice than BFS here? For as large a dimension as your system can handle, generate a plot of 'obstacle density p' vs 'probability that S can be reached from G'.

The DFS algorithm commences with the generation of 2 ordered pairs. The pairs are generated randomly using the Random.randInt() function. The first ordered pair is considered the beginning cell and the second order pair is considered the goal cell. Additionally two lists are created: the fringe and visited. The fringe is represented in the form of a stack. Visited represents the cells that were explored (all possible valid neighboring cells were added to the fringe). A cell is considered valid if that particular cell is open and that cell is not in both the fringe and visited. The algorithm first checks whether the beginning cell or the goal cell is closed. If either are closed the algorithm returns False. It then checks whether the beginning cell is equal to the goal cell. If they are equal, the algorithm returns True. If both beginning and goal cells are open and they are not equal to each other, the beginning cell is added to the fringe.

Code:

```
#checks whether either the goal or beginning points are blocked, if so return false
if self.tracking_array[int(beginning[0])][int(beginning[1])]==1 or self.tracking_array[goal[0]][
    goal[1]]==1:
    return False

#If they are the same point then return true
if beginning==goal:
    return True

#If not false, then add the beginning point to the fringe
self.fringe.append((int(beginning[0]) ,int(beginning[1])))
```

Then the algorithm enters a loop where it will iterate through the fringe, as long as the fringe length is greater than zero or a terminating condition is met. Inside the loop, the current cell is popped off the stack. The algorithm first checks whether the current cell is equal to the goal cell. If that is true then True is returned and the loop is exited (terminating case).

Code:

```
#loops through the fringe
while len(self.fringe) > 0:
    #sets current to the topmost element of the fringe
    current = self.fringe.pop()
```

```
    #Terminating case in which current is equal to the goal
    if current == (goal[0],goal[1]):
        return True
```

If the terminating case is false and the current cell is not in visited, the algorithm first checks whether the column of the current cell is nonzero. If it is nonzero, then there are 4 possible situations. The first situation checks the validity of the left child. If the left child is valid, the left child is added to the fringe. The second situation checks whether the row of the current cell is not equal to the last row and also checks the validity of the bottom child. If both conditions are met, then the bottom child is added to the fringe. The third situation checks whether the row of the current cell is not equal to the last column and also checks the validity of the right child. If both conditions are met then the right child is added to the fringe. The last situation checks whether the row of the current cell is not equal to zero and also checks the validity of the top child. If both conditions are met, then the top child is added to the fringe.

Code:

```
#current has not been explored yet (haven't added surrounding valid children to the fringe)
if current not in self.visited:
    #All columns other than the first column
    if current[1]>0:

        #Checks validity of left child
        if self.tracking_array[current[0]][current[1]-1]==0 and (current[0],current[1]-1) not in
            self.fringe and (current[0],current[1]-1) not in self.visited:
            self.fringe.append((current[0],current[1]-1)) #left child is valid

        #Checks whether the row is not the last row and also validity of bottom child
        if current[0]!=self.dim-1 and self.tracking_array[current[0]+1][current[1]]==0 and (current
            [0]+1,current[1]) not in self.fringe and (current[0]+1,current[1]) not in self.visited:
            self.fringe.append((current[0]+1,current[1])) #bottom child is valid

        #Checks whether the column is not the last column and validity of right child
        if current[1]!=self.dim-1 and self.tracking_array[current[0]][current[1]+1]==0 and (current
            [0],current[1]+1) not in self.fringe and (current[0],current[1]+1) not in self.visited:
            self.fringe.append((current[0],current[1]+1)) #right child is valid

        #Checks whether the row is not the first row and validity of top child
        if current[0]!=0 and self.tracking_array[current[0]-1][current[1]]==0 and (current[0]-1,
            current[1]) not in self.fringe and (current[0]-1,current[1]) not in self.visited:
            self.fringe.append((current[0]-1,current[1])) #top child is valid
```

In the case that the column of the current cell is zero, then there are 3 situations. The first situation checks whether the row of the current cell is not equal to the last row and also checks the validity of the bottom cell. If both conditions are met, then the bottom child is added to the fringe. The second situation checks the validity of the right child. If the right child is valid, then it is added to the fringe. The third situation checks whether the row is not equal to zero and also checks the validity of the top child. If both conditions are met, then the top child is added to the fringe. If the fringe has been totally iterated through and has not returned then False is returned as there is no possible path.

Code:

```
#Checks whether the row is not the last row and also validity of bottom child
if current[0]!=self.dim-1 and self.tracking_array[current[0]+1][current[1]]==0 and (current[0]+1,
    current[1]) not in self.fringe and (current[0]+1,current[1]) not in self.visited:
    self.fringe.append((current[0]+1,current[1])) #bottom child is valid
```

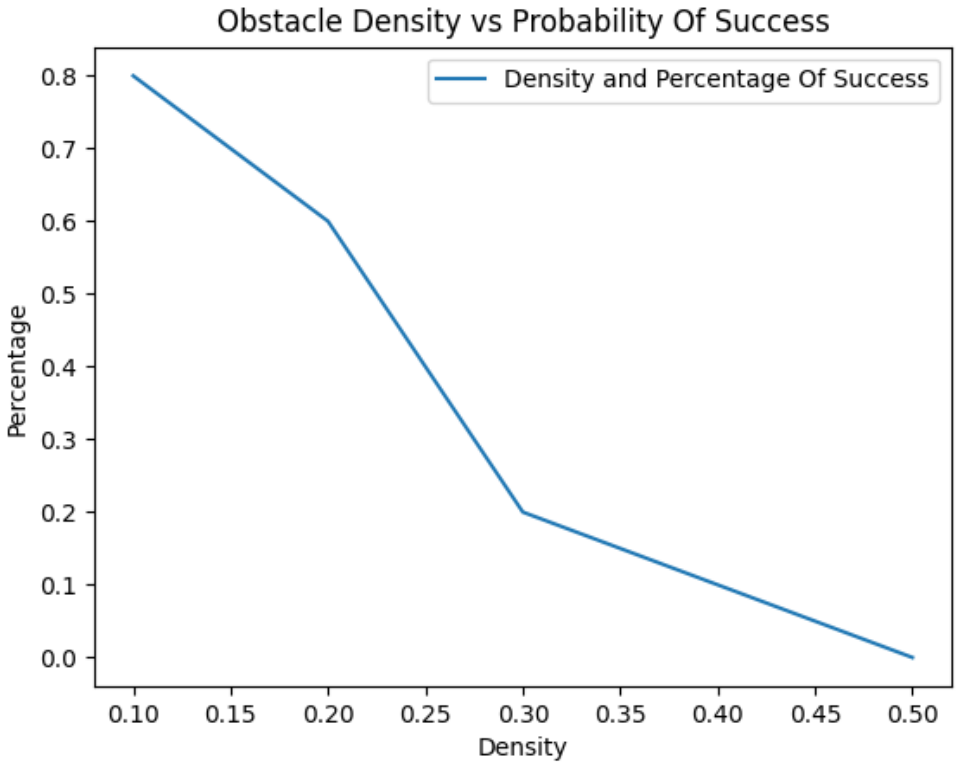
```
#Checks validity of right child
if self.tracking_array[current[0]][current[1]+1]==0 and (current[0],current[1]+1) not in self.
    fringe and (current[0],current[1]+1) not in self.visited:
   self.fringe.append((current[0],current[1]+1)) #right child is valid

#Checks whether the row is not the first row and validity of top child
if current[0]!=0 and self.tracking_array[current[0]-1][current[1]]==0 and (current[0]-1,current[1])
     not in self.fringe and (current[0]-1,current[1]) not in self.visited:
   self.fringe.append((current[0]-1,current[1])) #top child is valid

#Adds the current node to visited (all valid children have been added to the fringe)
self.visited.append(current)

#In the case that the fringe is empty and you could not find a path
return False
```

Diagram:



This dfs plot describes the relationship between obstacle density and rate of success. For each obstacle density, ten dfs tests were run to find the average rate of success : (number of success tests/10). As the obstacle density increased, the average rate of success decreased. This tests were done with a dimension of 250 and obstacle densities of 0.1-0.5.

DFS is better than BFS in this situation because the start and goal cells are on different levels. In DFS, the algorithm goes down a particular path until either the goal cell has been found or it has reached a dead end. In the case that it reaches a dead end, the algorithm backtracks and goes down the next path. In BFS, the algorithm goes level by level inspecting the all the cells on a particular level. So in the case that the goal node is deep down a path, DFS will find the node faster than BFS. BFS works better when start and goals cells are close by (level wise).

3.3 Write BFS and A* algorithms (using the euclidean distance metric) that take a maze and determine the shortest path from S to G if one exists. For as large a dimension as your system can handle, generate a plot of the average 'number of nodes explored by BFS - number of nodes explored by A*' vs 'obstacle density p'. If there is no path from S to G, what should this difference be?

The BFS algorithm follows the structure of a tree search where S is the coordinate (0, 0) and G is the coordinate (dimension-1, dimension-1) where S is stored into a fringe represented by a queue. Then before starting the BFS process we need to create a visited array that will hold the positions that have been visited (True) and not visited (False). Also we need a path array that holds the parent's position to backtrack once we find the goal node.

Code:

```
...
arr = self.tracking_obstacles

# now define the start and end node which in our case is the first indicies and the last indicies
    respectively
start = (0, 0)
goal = (len(arr) - 1, len(arr) - 1)

# now because we are working with bfs, we know bfs calls for a fringe in the form of a queue
    because of the queue's policy (FIFO)
fringe = deque()
fringe.append(start)

# keep an array to represent the visited arrays
visited = numpy.zeros((len(arr), len(arr)), dtype=bool)

# for this implementation of bfs we want to keep track of the parents to obtain the shortest path
path = []
...
```

Then while iterating through the queue, the current position is popped from the queue, marked as visited and checked if it equal G. Before adding the neighbors to the queue.

Code:

```
...
# now iterate through the fringe to check for the path
while len(fringe) > 0:
    current = fringe.popleft()
    visited[current[0]][current[1]] = True
    if current == goal:
        # backtracks to start node (will be explained later)
    else:
    # ...
...
```

If the current position that is popped is not equal to the goal node then the algorithm will search up, down, left, right of the current cell (explore its neighbors) then add it to the fringe if it is not an obstacle and the direction its checking is valid. Then if the position wasn't previously visited and the position is not already in the fringe. If that condition is true it will also store the parent into another dictionary holding the parents.

Code:

```
...
# first check the up direction
if self.check_valid_bounds(-1, 0, current, arr) and arr[current[0] - 1][current[1]] == 0 and
    visited[current[0] - 1][current[1]] == False and (current[0] - 1, current[1]) not in fringe:
       fringe.append((current[0] - 1, current[1]))
       if current not in path: # only append the parent if its not seen in the path already
           path.append(current)

       # now check the down direction
       if self.check_valid_bounds(1, 0, current, arr) and arr[current[0] + 1][current[1]] == 0 and
           visited[current[0] + 1][current[1]] == False and (current[0] + 1, current[1]) not in
           fringe:
           fringe.append((current[0] + 1, current[1]))
           if current not in path: # only append the parent if its not seen in the path already
               path.append(current)

       # now we can check the left direction
       if self.check_valid_bounds(0, -1, current, arr) and arr[current[0]][current[1] - 1] == 0 and
            visited[current[0]][current[1] - 1] == False and (current[0], current[1] - 1) not in
           fringe:
           fringe.append((current[0], current[1] - 1))
           if current not in path: # only append the parent if its not seen in the path already
               path.append(current)

       # finally check the right side
       if self.check_valid_bounds(0, 1, current, arr) and arr[current[0]][current[1] + 1] == 0 and
           visited[current[0]][current[1] + 1] == False and (current[0], current[1] + 1) not in
           fringe:
           fringe.append((current[0], current[1] + 1))
           if current not in path: # only append the parent if its not seen in the path already
               path.append(current)
...
```

When the condition of the current position being G is true, the algorithm will back track from the final node until it reaches the starting node and return the path it found. Then return that path for the results and draw it for the visualization.

Code:

```
if current == goal:
    path.append(current)
    self.bfs_nodes = len(path)
    path.reverse()
    # now that we found the end node, let's perform a recursive backtracking algorithm to find the
        actual path
    bfs_route = []
    while path[0] != start:
        new_curr = path.pop(0)
        if not bfs_route:
            bfs_route.append(new_curr)
```

```
        # check if its a straight path up
        elif new_curr[1] == bfs_route[len(bfs_route) - 1][1] + 1 and new_curr[0] == bfs_route[len(
            bfs_route) - 1][0]:
            bfs_route.append(new_curr)
        # check if its a straight path right
        elif new_curr[1] == bfs_route[len(bfs_route) - 1][1] and new_curr[0] == bfs_route[len(
            bfs_route) - 1][0] + 1:
            bfs_route.append(new_curr)
        # check if its a straight path down
        elif new_curr[1] == bfs_route[len(bfs_route) - 1][1] - 1 and new_curr[0] == bfs_route[len(
            bfs_route) - 1][0]:
            bfs_route.append(new_curr)
        # check if its a straight path left
        elif new_curr[1] == bfs_route[len(bfs_route) - 1][1] and new_curr[0] == bfs_route[len(
            bfs_route) - 1][0] - 1:
            bfs_route.append(new_curr)

    # append the last node because that won't be included (we check until the starting node)
    bfs_route.append(start)
    bfs_route.reverse()
    self.draw_path(bfs_route) # once the final path is recieved, draw it for the visualization
    return bfs_route
```

The A* algorithm is a way to do an informed search. The search relies on a heuristic value. The heuristic value is the estimate cost of getting from the current location to goal. An admissible heuristic value should be equal or lower than the actual cost of getting from current location to goal. In the A* implementation for this project the heuristic is calculated by finding the euclidean distance from the current location to goal. In addition there is also a 2D array that keeps track of the current node's parent by the numbers it holds.

Code:

```
    def a_star(self): # A* algo
        maze_array = self.tracking_obstacles
        fringe = [] # priority queue
        visited = [[-1, -1, -1]] # keeps track of all the visited cells
        child1 = []
        child2 = []
        child3 = []
        child4 = []
        n = len(maze_array)
        start = [0, 0]
        cost = numpy.zeros([n, n])
        goal = [n - 1, n - 1]
        tracker = [] # array for final path
        fringe.append(start)
        parent = numpy.zeros([n, n]) # 3 top, 4 right, 1 bottom, 2 left - to keep track of the
            parent of each node
...
```

Another required part for this search is the priority queue that stores all the nodes to be explored next. The nodes are ordered by calculating the cost to get to that point and then adding the heuristic value. The node that has the lowest number when the cost to get there is added to the heuristic value is kept in the front of the queue. In every iteration the front node in priority queue is popped and its neighbours are explored. This goes on until the goal is found or there is no more nodes in the priority queue meaning there is no path from start to goal.The code under is inside a loop that keeps running till there is a node in fringe or the goal node is found.

Code:

```
...
        while len(fringe) > 0:
            current = fringe.pop(0) # take out the child with highest priority
            if len(child1) != 0: # empty the child arrays
                child1.pop()
            if len(child2) != 0:
                child2.pop()
            if len(child3) != 0:
                child3.pop()
            if len(child4) != 0:
                child4.pop()
            if current not in visited: # only continue if the current node is not visited before
                if not fringe: # if the fringe is empty it does not check for child in fringe
                    if current[0] != 0 and maze_array[current[0] - 1][current[1]] != 1: # checks if
                        it's not the top row and that there is not an obstacle on top of it
                        child1.append([current[0] - 1, current[1]]) # top cell
                        if child1[0] not in visited and cost[current[0] - 1][[current[1]]] >= cost[
                            current[0]][ # if the child is not visited
                            [current[1]]] + 1 or cost[current[0] - 1][[current[1]]] == 0: # and the
                                path it took is shorter than beforeee or it's the firt time getting
                                there
                            cost[current[0] - 1][[current[1]]] = cost[current[0]][[current[1]]] + 1 #
                                then add it to the fringe with priority queue
                            fringe = self.sorting(fringe, child1, cost)
                            parent[current[0] - 1][[current[1]]] = 3 # set its parent to the top cell
                    if current[1] != n - 1 and maze_array[current[0]][current[1] + 1] != 1:
                        child2.append([current[0], current[1] + 1]) # right cell
                        if child2[0] not in visited and cost[current[0]][[current[1] + 1]] >= cost[
                            current[0]][
                            [current[1]]] + 1 or cost[current[0]][[current[1] + 1]] == 0:
                            cost[current[0]][[current[1] + 1]] = cost[current[0]][[current[1]]] + 1
                            fringe = self.sorting(fringe, child2, cost)
                            parent[current[0]][[current[1] + 1]] = 2
                    if current[0] != n - 1 and maze_array[current[0] + 1][current[1]] != 1:
                        child3.append([current[0] + 1, current[1]]) # bottom cell
                        if child3[0] not in visited and cost[current[0] + 1][[current[1]]] >= cost[
                            current[0]][
                            [current[1]]] + 1 or cost[current[0] + 1][[current[1]]] == 0:
                            cost[current[0] + 1][[current[1]]] = cost[current[0]][[current[1]]] + 1
                            fringe = self.sorting(fringe, child3, cost)
                            parent[current[0] + 1][[current[1]]] = 1
                    if current[1] != 0 and maze_array[current[0]][current[1] - 1] != 1:
                        child4.append([current[0], current[1] - 1]) # left cell
                        if child4[0] not in visited and cost[current[0]][[current[1] - 1]] >= cost[
                            current[0]][
                            [current[1]]] + 1 or cost[current[0]][[current[1] - 1]] == 0:
                            cost[current[0]][[current[1] - 1]] = cost[current[0]][[current[1]]] + 1
                            fringe = self.sorting(fringe, child4, cost)
                            parent[current[0]][[current[1] - 1]] = 4
...
```

Here if the fringe is not empty we can check through the fringe for current code and any neighbour of the current we want to be explored and added to the fringe. We only add nodes that are not already visited and not in the fringe. We also check if the cost to get to a neighbour is smaller then it was before, and if it is, the distance is updated and the neighbour added to the fringe.

Code:

```
            else:
                if current not in fringe: # if current is not in fringe we go through its
                    neighbours
                    if current[0] != 0 and maze_array[current[0] - 1][current[1]] != 1:
                        child1.append([current[0] - 1, current[1]]) # top cell
                        if child1[0] not in visited and child1[0] not in fringe and cost[current
                            [0] - 1][
                            [current[1]]] >= cost[current[0]][[current[1]]] + 1 or cost[current[0]
                                - 1][ # if the child is not visited before and not in the fringe
                                and there is no obstacle there
                            [current[1]]] == 0: # if the cost to get there is less than before or
                                it's 0 - meaning it's first time getting there
                            cost[current[0] - 1][[current[1]]] = cost[current[0]][[current[1]]] +
                                1 # set cost to get to child: cost ot get to current + 1
                            fringe = self.sorting(fringe, child1, cost) # add child to the fringe
                                - priority queue
                            parent[current[0] - 1][[current[1]]] = 3 # set it's parent to top cell
                    if current[1] != n - 1 and maze_array[current[0]][current[1] + 1] != 1:
                        child2.append([current[0], current[1] + 1]) # right cell
                        if child2[0] not in visited and child2[0] not in fringe and cost [current
                            [0]][
                            [current[1] + 1]] >= cost[current[0]][[current[1]]] + 1 or cost[
                                current[0]][
                            [current[1] + 1]] == 0:
                            cost[current[0]][[current[1] + 1]] = cost[current[0]][[current[1]]] +
                                1
                            fringe = self.sorting(fringe, child2, cost)
                            parent[current[0]][[current[1] + 1]] = 2
                    if current[0] != n - 1 and maze_array[current[0] + 1][current[1]] != 1:
                        child3.append([current[0] + 1, current[1]]) # bottom cell
                        if child3[0] not in visited and child3[0] not in fringe and cost[current
                            [0] + 1][
                            [current[1]]] >= cost[current[0]][[current[1]]] + 1 or cost[current[0]
                                + 1][
                            [current[1]]] == 0:
                            cost[current[0] + 1][[current[1]]] = cost[current[0]][[current[1]]] +
                                1
                            fringe = self.sorting(fringe, child3, cost)
                            parent[current[0] + 1][[current[1]]] = 1
                    if current[1] != 0 and maze_array[current[0]][current[1] - 1] != 1:
                        child4.append([current[0], current[1] - 1]) # left cell
                        if child4[0] not in visited and child4[0] not in fringe and cost[current
                            [0]][
                            [current[1] - 1]] >= cost[current[0]][[current[1]]] + 1 or cost[
                                current[0]][
                            [current[1] - 1]] == 0:
                            cost[current[0]][[current[1] - 1]] = cost[current[0]][[current[1]]] +
                                1
                            fringe = self.sorting(fringe, child4, cost)
                            parent[current[0]][[current[1] - 1]] = 4
                visited.append(current)
```

If the current goal is the goal node, the parent array is then used to backtrack the path from goal to start and return the nodes for the path. Each value of the cell in parent array represents which neighbour is it's parent, the value of 1 means that the parent is it's top cell, value of 2 means that the parent is it's right cell, value of 3 means the parent is it's bottom cell and value of 4 means the parents is it's left cell. Each parent is added to the tracker array and it is reversed to get the path from start to goal and then returned for graphing.

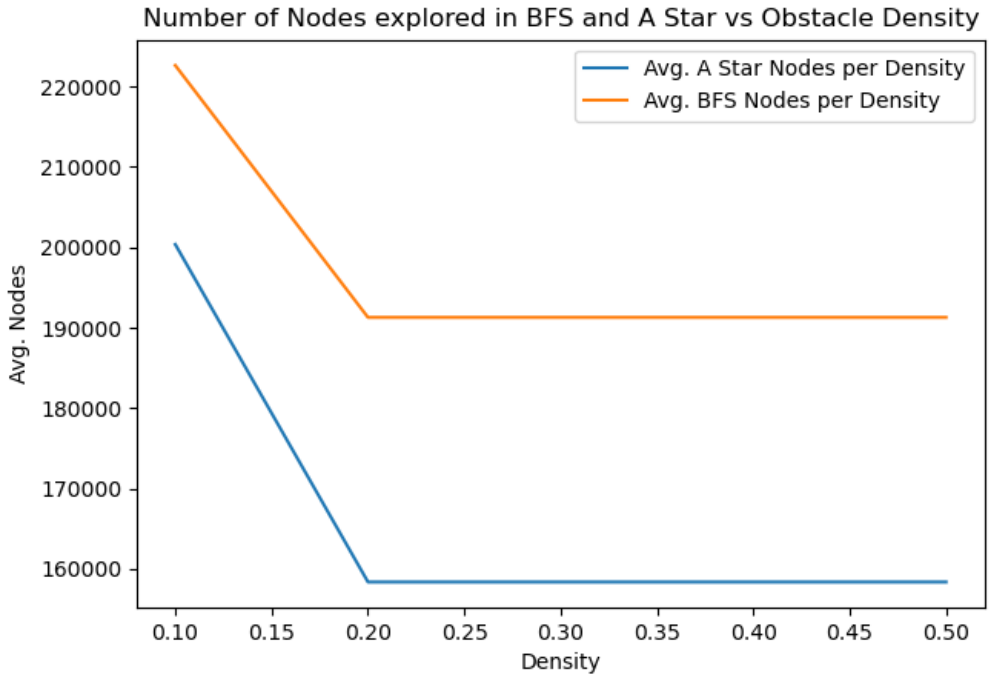
Code:

```
            if current == goal: # takes the "parent" array and tracks back to the start using the
                cell value
                y = n - 1
                x = n - 1
                tracker.append([y, x])
                while True:
                    if parent[y][x] == 1: # parent is top cell
                        tracker.append([y - 1, x])
                        y -= 1
                    elif parent[y][x] == 2: # parent is right cell
                        tracker.append([y, x - 1])
                        x -= 1
                    elif parent[y][x] == 3: # parent is bottom cell
                        tracker.append([y + 1, x])
                        y += 1
                    elif parent[y][x] == 4: # parent is left cell
                        tracker.append([y, x + 1])
                        x += 1
                    if x == 0 and y == 0: # when it reaches start it breaks out of the loop
                        break
                tracker.reverse()
                self.draw_path(tracker) # draws the path
                return True

        return False
```

There are many characteristics that make BFS and A* different. Most notably A* is more of an informed approach that uses heuristic that tells us the Euclidean distance a cell is from the goal, while BFS is uninformed and not aware of things like the cost it takes to take a certain route. Now comparing the algorithms side by side we conducted tests for various obstacles densities at 0.1, 0.2, 0.3, 0.4 and 0.5 on a maze with a 500x500 size where we compared the number of nodes explored by each algorithm.

Results:



As we would expect, BFS will end up exploring more nodes on average because when the algorithm calls for exploring all the neighbors of the node that was popped from the fringe in an uninformed method, the fringe size will be exponential and it will create a space complexity of $O(2^n)$. This is due to the fact that it adds every single neighbor that it explorers, if it's not an obstacle, to the fringe and explores them in an uninformed way. On the other hand, A* will explore less nodes than BFS because A* applies more of an informed approach when exploring the nodes. A* relies on an heuristic that is the euclidean distance it is away from the agent. A* compared to BFS would only have a space complexity of $O(n)$ where n represents the amount of spots in the maze. Based on this search it won't need to explore unnecessary nodes like BFS would do because of its uninformed nature. Now for the scenario when there is no path from S to G in the maze, there would be no difference between the number of nodes explored with BFS and A*. Both the algorithms will explore every reachable node from the start for the case where there is no path. This makes sense since the algorithms would have to explore every node to come to a decision that there is no path from start to goal.

3.4 What's the largest dimension you can solve using DFS at p = 0.3 in less than a minute? What's the largest dimension you can solve using BFS at p = 0.3 in less than a minute? What's the largest dimension you can solve using A at p = 0.3 in less than a minute?

The largest dimension that can be solved using DFS at p= 0.3 in less than a minute is 250. This is done with start and goal being randomly generated. However, When solving with a dimension of 300 with points (0,0) and (299,299) it also finishes in less than a minute. The reason that randomly generated coordinate pairs execute slower is that the start and goal cells could be close by but due to the nature of DFS, the goal cell is not popped from the fringe until close to the end of the execution. For example if the current cell is (0,0) and the goal cell is (1,0) and the cells (0,1) and (1,0) are open, DFS would first add (1,0) to the fringe and then (0,1). The algorithm would explore (0,1) (moving down that path) before exploring (1,0). If (0,1)'s path is large, (1,0) will be explored at the end of the execution, hence the large runtime.

The largest dimension maze, BFS can solving at a obstacle density of $p = 0.3$ in less than a minute is a 370x370 maze which takes 57.36913275718689 seconds to solve. Based on some observations when conducting the experiments to test for the possible largest BFS we can run it appears to be that the lower dimension maze sizes ran a lot quicker than a dimension ≥ 101. can be solved in anytime $0<x<1$ where x represents the amount of seconds it takes to solve the maze. Then anything larger than 1, adds alot more computational time and space complexity where the fringe can exponentially increase so seeing times >1 second is more reasonable. The more area in the maze, would equate to more cells BFS would have to discover before finding the shortest path. The one main reasonable approach to still using an algorithm like BFS would be the optimality.

As for A*, the maximum maze size that can be solved under a minute is 180x180, which took 59.9 seconds. This time can be brought down by optimizing the code to take up less computation for searching when it is looking for the cells in "visited" and "fringe" array. In addition, the time to compute the path also depends on the maze itself. For example if there are a lot of obstacles in the middle it will take more time to find a path than if the obstacles were mostly on the sides.

3.5 Describe your improved Strategy 3. How does it account for the unknown future?

For strategy 3, the main way we decided to create an improved strategy was by first seeing what the previous strategies (1 and 2) accounted for and what could be improved from those strategies. Strategy 1 for instance accounts for if the agent catches on fire but not the future states of the fire. While strategy 2 accounts for both if the agent catches on fire and future states but doesn't account for the future states beyond one step. The future in this context being the chance of the other cells catching on fire based on the current cells that are on fire. To generate a fire, there was a formula that was applied to indicate a fire on a certain cell, so using that we thought of creating a Node class that would represent the probability of a cell catching on fire and the value (0 = empty cell, 1 = obstacle and 2 = fire).

Code:

```
class FireNode:
    fire_prob = 0.0 # indicates the probability of a cell to catch on fire
    value = 0 # indicates what the cell is (open, fire, or obstacle)

    def __init__(self, value, prob_fire):
        self.fire_prob = prob_fire
        self.value = value
```

Now that we can represent the maze using this Node class upon every fire generation we want to create another instance of the Maze that will represent the states of the fire, obstacles and free spaces but also the probability of a certain cell catching on fire. So upon generating the fire we can create this representation. So if the fire generation starts we know that its going to find a random spot to start the fire so we can create a node there.

Code:

```
if fire_maze[x][y] != 2 and fire_maze[x][y] != 1 and (x != 0 and y != 0) and (x != len(fire_maze) -
     1 and y != len(fire_maze) - 1): # if a spot is already not on fire and not the first or last
    cell make it on fire
    fire_array[x][y] = 2 # 2 indicates there is fire placed
    self.tracking_obstacles[x][y] = 2 # this is so we can take account for the maze globally
    self.fire_index += 1 # increase this so we dont choose another random spot
    self.fire_maze[x][y] = FireNode(2, 1.0) # Fire Node is 2 and 1 (100%) of catching on fire
        because its on fire already
```

```
    return self.tracking_obstacles
```

After we create the one fire starting position we want to track the rest of the fire positions, obstacle position and free cells in the fire_maze field so we can utilize it during tracking the shortest path later on. So while iterating through each cell of the maze if there isn't any obstacle or fire placed in the maze, calculate the number of surrounding cells that are on fire by incrementing a fire variable that will add one if a neighbor of a cell is on fire. Then calculating a probability to make the current cell on fire by using a formula $1-(1-q)^k$ where q is the likely hood of a cell catching on fire and k (fire) is the number of neighbors that are currently on fire.

Code:

```
if fire_maze[i][j] != 1 and fire_array[i][j] != 2:
    if i != len(self.tracking_obstacles) - 1 and fire_array_copy[i + 1][j] == 2: # check the below
        neighbor
        fire += 1
    if fire_array_copy[i - 1][j] == 2 and i != 0: # check the up neighbor
        fire += 1
    if j != len(self.tracking_obstacles) - 1 and fire_array_copy[i][j + 1] == 2: # check the right
        neighbor
        fire += 1
    if fire_array_copy[i][j - 1] == 2 and j != 0: # check the left neighbor
        fire += 1
    prob = 1 - ((1 - q) ** fire) # calculate the probability of a cell catching on fire
```

Once you calculate that probability, a cell will only catch on fire if the number of neighbors that are on fire isn't 0, the calculated probability isn't 0 and if a randomly generated number is <the calculated probability.

Code:

```
if fire > 0 and random.random() <= prob and prob > 0:
    fire_array[i][j] = 2 # track it locally
    self.tracking_obstacles[i][j] = 2 # track it globally
    self.fire_maze[i][j] = FireNode(2, 1.0) # Fire Node is 2 and 1 (100%) of catching on fire
        because its on fire already
```

Otherwise, if the cell can't catch on fire we know that the cell is still empty so we can calculate the probability of the node catching on fire by taking the same probability and adding that node to the cell with a value of 0 to indicate that its still an empty cell.

Code:

```
else:
    # indicate the cell has a certain chance of catching on fire if its not based on the variable
        prob
    self.fire_maze[i][j] = FireNode(0, prob)
```

Now for any obstacle, the chance of a cell catching on fire is automatically 0 because there is no chance of the fire spreading to an obstacle so:

Code:

```
elif fire_maze[i][j] == 1: # there is an obstacle
    self.fire_maze[i][j] = FireNode(1, 0.0) # obstacles would be indicated with 1 and have no
        chance of catching on fire
```

After we are able to create a global representation of the maze that is on fire comes the logic behind finding the path. Now for finding the path we applied a slightly modified version of BFS. Before BFS was more of an uninformed search where it would take any neighbor however we want to apply an informed approach now because we have this representation of the maze that knows the probability of a cell catching on fire and the states that are already on fire. So anytime we take a neighbor into the fringe for BFS we want to see if the probability of the cell catching on fire is lower than 30% because if you were to take a node with higher than 30% there is a higher chance of other nodes that are on fire to make that cell on fire. While if its already a lower percentage you have a chance that the node is a good choice. Then also checking for the current states of the fire, we would check if any direction of the maze is not on fire which we will also add that too the fringe.

Code:

```
else:
    # first check the up direction now checking the probability of cell on fire is less than .3
    if self.check_valid_bounds(-1, 0, current, arr) and arr[current[0] - 1][current[1]] == 0 and
        self.fire_maze[current[0] - 1][current[1]].fire_prob < .3 and visited[current[0] - 1][
        current[1]] == False and (current[0] - 1, current[1]) not in fringe:
        fringe.append((current[0] - 1, current[1]))
        if current not in path:
            path.append(current)

    # now check the down direction now checking the probability of cell on fire is less than .3
    if self.check_valid_bounds(1, 0, current, arr) and arr[current[0] + 1][current[1]] == 0 and
        self.fire_maze[current[0] + 1][current[1]].fire_prob < .3 and visited[current[0] + 1][
        current[1]] == False and (current[0] + 1, current[1]) not in fringe:
        fringe.append((current[0] + 1, current[1]))
        if current not in path:
            path.append(current)

    # now we can check the left direction now checking the probability of cell on fire is less than
        .3
    if self.check_valid_bounds(0, -1, current, arr) and arr[current[0]][current[1] - 1] == 0 and
        self.fire_maze[current[0]][current[1] - 1].fire_prob < .3 and visited[current[0]][current
        [1] - 1] == False and (current[0], current[1] - 1) not in fringe:
        fringe.append((current[0], current[1] - 1))
        if current not in path:
            path.append(current)

    # finally check the right side now checking the probability of cell on fire is less than .3
    if self.check_valid_bounds(0, 1, current, arr) and arr[current[0]][current[1] + 1] == 0 and
        self.fire_maze[current[0]][current[1] + 1].fire_prob < .3 and visited[current[0]][current
        [1] + 1] == False and (current[0], current[1] + 1) not in fringe:
        fringe.append((current[0], current[1] + 1))
        if current not in path:
            path.append(current)
```

Now that we modified BFS to take a more informed approach now comes for how we would execute the strategy. For some key points to think about when implementing strategy 3 we had to remember:

1. The fire is constantly moving
2. There is a point where the agent is going to die or be alive
3. Only draw one position at a time and we would have to change the modified BFS code to take a variable starting position depending on the position we are drawing.
4. if the path returned is [ ] that would mean the agent died or there is no path so we can immediately exit

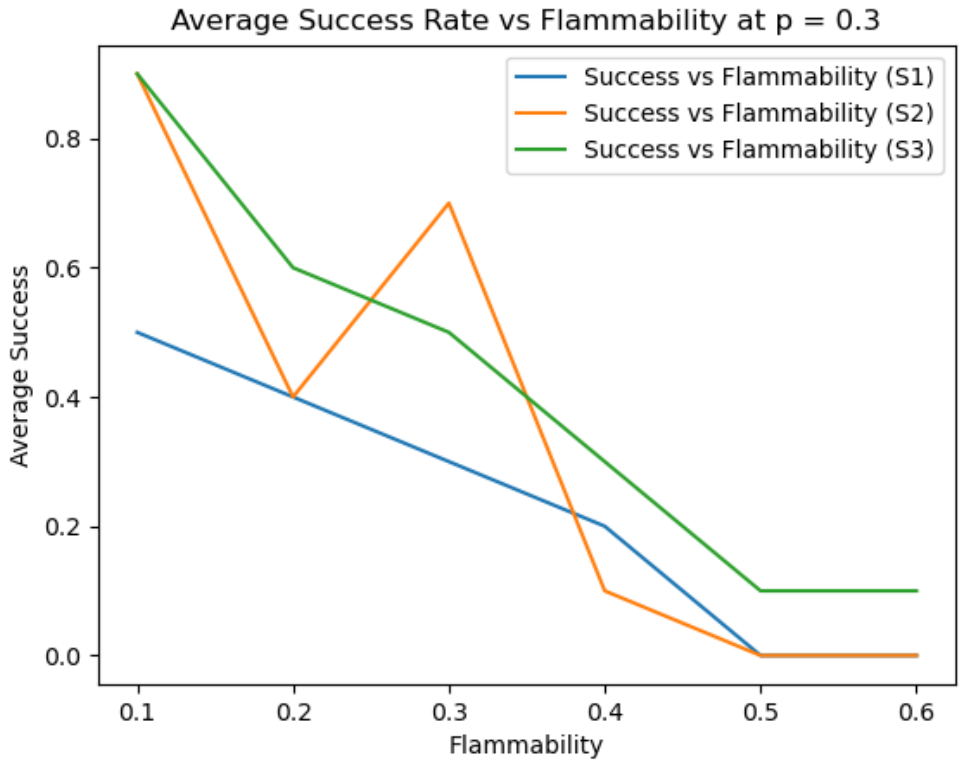
So while strategy 3 would run infinitely until a certain point where the agent escaped or turns out dead which we will indicate with the variable ALIVE. In every iteration, we want to keep generating the fire and sleep so the route calculation has some time to catch up. Then at every step (similar to strategy 2) calculate the shortest path using the modified BFS to account for the probability of a cell catching on fire. Then if that path returns [ ] return ALIVE as False breaking out of the loop, then returning ALIVE to indicate the agent died. Otherwise draw out the path until the agent gets to the goal node and at the end return True to indicate the agent survived.

Code:

```
# this is the name of strategy 3: escape the fire
def cat(self):
    ALIVE = True # indicates if the agent is alive which it will be when it starts
    start = (0, 0) # we want to start in the beginning of the maze
    # keep going until ALIVE turns to False (indicating the agent died or no path) or if the agent
        made it through
    while ALIVE:
        self.generate_fire_maze(float(sys.argv[4])) # generate the fire at a given rate based on the
            command line
        time.sleep(1) # for calculation
        escape_route = self.fire_route_search(start)
        if len(escape_route) == 0: # indicates the agent died
            ALIVE = False
            break
        elif escape_route[0] == (self.dim-1, self.dim-1): # indicates the agent made it through
            break
        # keep drawing the agent going through the maze
        self.draw_path(escape_route[1])
        start = escape_route[1] # because we are drawing one path at a time make start the next
            position

    return ALIVE
```

3.6 Plot, for Strategy 1, 2, and 3, a graph of average strategy success rate vs flammability q at p = 0.3. Where do the different strategies perform the same? Where do they perform differently? Why?



For each strategy 10 tests were run for flammability rate "q" ranging from .1 to .6 with increments of .1 and the average success rate was calculated based on how many time the agent made it to the goal without burning in the fire. During the testing, a test where there was no path from the start to goal or start to the first fire was not counted in calculating the average success rate. As seen in the plots the average success rate is very similar when the flammability rate "q" is high, but at smaller "q" values, the success rate differs. The difference between the success rate higher when the flammability rate "q" is low. The similarity in the success rate in all 3 strategies when the flammability rate "q" is high can be attributed to the fact that fire is spreading quickly and it has a higher chance of blocking the agents path before even if the agents tries to avoid it. Since we only ran 10 tests it is hard to see the actual difference between the success rate of the strategies, but if we ran more tests we would be able to solidify the difference between the strategies more.

3.7 If you had unlimited computational resources at your disposal, how could you improve on Strategy 3?

Pseudocode:

```
def ultimate_power():
    maze = build_maze() # builds the maze that contains 0 = free cell, 1 = obstacle, 2 = fire
    Status = True # to see if the agent is alive or not
    fire_distance = fire_distance() # keeps track of the distance from agent to the fire nodes
    copy_fire_array = fire_probability(maze) # this this is the probability distribution of where
        the fire can go in the next n/10 steps, where n is maze dimension
    fringe = [] # priority queue fringe
    next_step = [0,0]
    if there is a path from start to goal:

        while status and there is a path from next_step to goal :
            maze.update() # fire moves one step
            copy_fire_array = fire_probability(maze) # update the probability distribution for next
                n/10 steps
            fire_distance = fire_distance() # updates the fire distances

            next_step = A_star(maze) # run A* on the updated maze, the heuristic used for this is
                the probability distribution of the fire for next n/10 steps and the distance from
                the closest fire spot, where n is the maze size

            if next_step is empty: # if there is no more steps to be taken then return true
                return Status
            elif next_step: # if the agent went into fire return false
                return False
```

Based on the current Strategy 3, we keep track of the probabilities of a cell to catch on fire and the current state of the fire while using BFS to find the path to the end however if we were in the situation where we had unlimited computation sources at our disposal we can utilize A*'s power of keeping a heuristic of the Euclidean distance the cell is away from the nodes that are on fire, the probability that a cell has to catch on fire and the current states of the fire. With this improvement we can utilize this new heuristic to stay completely away from the fire while still accounting for the future states of the fire using the probability of cell to catch on fire. Also keeping track of the current states of the fire all at once. Which would make Strategy 3 account for all the possibilities the fire might go and get the agent to the other side. Unless the ending is blocked by fire or the obstacles at hand.

3.8 If you could only take ten seconds between moves (rather than doing as much computation as you like), how would that change your strategy? Describe such a potential Strategy 4.

Pseudocode:

```
...
 maze = build_maze() # builds the maze outside the main strategy that contains the global
     representation containing 0 = free cell, 1 = obstacle, 2 = fire, 3 = path
...
def ten_sec_moves():
    ALIVE = True # indicates if the agent is still alive
    while Alive:
        generate_fire(maze, flammability) # just generates fire on the given global tracking maze
            which is constantly updated
        delay_system(1) # delay by 1 second
```

```
        bfs = escape_route(maze, starting) # uses the global maze representation too find the route
            ONLY checking for if there is a 0 cell for each neighbor to take
        if bfs == [] or bfs[0] is fire:
            ALIVE = False
        elif bfs[0] = ending:
            break
        draw_path(bfs[1]) # agent moves one cell at a time
        starting = bfs[1]
    return ALIVE
```

If we could only take ten seconds between moves (rather than doing as much computation as we like) there would be significant changes to strategy 3. As of the current rendition of Strategy 3, we calculate all the positions of the maze and keeping track of the probability that the cell can catch on fire. Which takes a lot of computation time from each time the agent moves it would be calculating these factors on top of finding the optimal path. But if we were to only have 10 seconds, then the most optimal way to move around would be reducing complexity first by keeping track of one global maze with set values to represent an obstacle, free space and the fire. Then just checking in that global maze if the space is simply 0 then take that neighbor in the modified BFS. While this would be less informed than the modified BFS we created however this modified version would reduce time complexity. Which would be done now in $O(n)$ because the time complexity would be equivalent to normal BFS where the edges V and vertices E which we can just make n. Then the space complexity would still be $O(2^n)$ because the mazes are unknown sizes with unknown obstacle densities that are chosen at random based on the user.

4 Contributions and Acknowledgments about Rutgers Academic Policy

Group Member: Karun Kanda **Netid:** kk951 **RUID:** 184007619

For the final write up I worked on the parts that involved BFS and Strategy 3 because that's what I was in charge of for the code. Also, contributed to some of the write up for comparing A* and BFS.

I, Karun Kanda, certify that the work submitted in this project is my own work and not copied or taken from online or any other student's work.

Group Member: Varun Gaddam **Netid:** vrg24 **RUID:** 187008136

For the final write up I worked on the parts that involved DFS and Strategy 2 because that's what I was in charge of for the code.

I, Varun Gaddam, certify that the work submitted in this project is my own work and not copied or taken from online or any other student's work.

Group Member: Agam Modasiya **Netid:** ajm432 **RUID:** 185009911

For the final write up I worked on the parts that involved A* explanation and Strategy 1 because I was responsible for coding those parts. I also helped with getting test results for Strategy 1 and 3.

I, Agam Modasiya, certify that the work submitted in this project is my own work and not copied or taken from online or any other student's work.