CS440 Project 1: Maze on Fire

Achintya Singh, Josh Arleth, Tapan Patel

February 20, 2021

Exercise 1

Write an algorithm for generating a maze with a given dimension and obstacle density p

```
from matplotlib import pyplot
import random
import numpy as np


#takes maze dims and wall probability and returns a 2d array that
    represents a maze
def maze_generator(maze_dim, wall_prob, set_fire):

   #create array of zeros
   maze = np.zeros((maze_dim,maze_dim))

   #find how many walls we need then randomly set them
   for _ in range(int (wall_prob * maze_dim * maze_dim) ):
      rand_row = random.randint(0, maze_dim - 1)
      rand_col = random.randint(0, maze_dim - 1)

      #making sure we dont make the start or end a wall
      if (rand_row == 0 and rand_col == 0) or (rand_row == maze_dim - 1 and
          rand_col == maze_dim - 1):
         continue
      maze[rand_row][rand_col] = 50 #wall id could be anything, made it 50

   #if we want to light this maze on fire
   if set_fire:

      #try 100 times to find a suitable location for the fire
      #if 100 times isn't enough then something is wrong
      for _ in range(0, 100):
         rand_row = random.randint(0, maze_dim - 1)
         rand_col = random.randint(0, maze_dim - 1)

         #Making sure we aren't settin gthe starting or ending point on
            fire and it's not a wall
         if maze[rand_row][rand_col] != 50 and ( (rand_row != 0 and
```

```
            rand_col != 0) and (rand_row != maze_dim - 1 and rand_col !=
            maze_dim - 1) ):
           maze[rand_row][rand_col] = 75 #fire id could be anything, made
               it 75
           break
   return maze

#takes a 2d array representing a maze and prints as well as plots it
def print_mazes(maze_list):

   maze_id = 0
   for maze in maze_list:
      pyplot.figure(maze_id, figsize=(10,10))
      pyplot.imshow(maze)
      maze_id += 1
   pyplot.show()
```

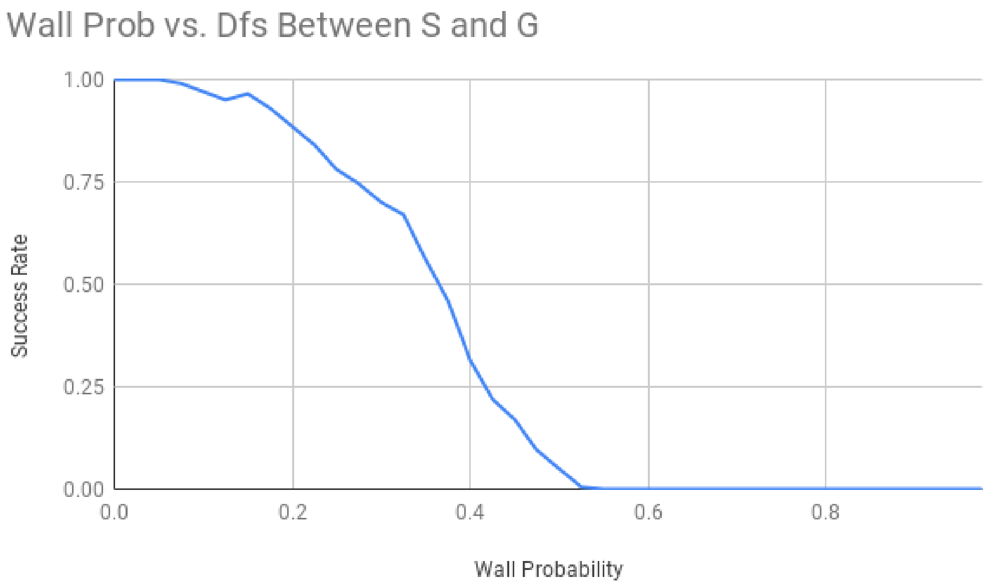
Exercise 2

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'.

```
from maze_generator import *
from point import *
from maze_functions import *


def dfs(start, goal, maze):

    nodes_exp = 0
    fringe = [] #using python list as a stack
    fringe.append(start)
    closed_matrix = np.zeros((maze.shape[0],maze.shape[0]))

    while fringe:

        cp = fringe.pop()

        nodes_exp += 1 #countin the number of nodes explored

        #checking if reached goal
        if is_goal(cp, goal):
            return cp, nodes_exp

        #generating children
        npt = point(cp.row, cp.col + 1, cp)
        if valid_point(maze, npt, 50, 75 ) and not in_fringe_stack(fringe,
            npt) and not visited(closed_matrix, npt):
            closed_matrix[npt.row][npt.col] = 1
            fringe.append(npt)

        npt = point(cp.row, cp.col - 1, cp)
        if valid_point(maze, npt, 50, 75 ) and not in_fringe_stack(fringe,
            npt) and not visited(closed_matrix, npt):
            closed_matrix[npt.row][npt.col] = 1
            fringe.append(npt)

        npt = point(cp.row + 1, cp.col, cp)
        if valid_point(maze, npt , 50, 75 ) and not in_fringe_stack(fringe,
            npt) and not visited(closed_matrix, npt):
            closed_matrix[npt.row][npt.col] = 1
            fringe.append(npt)

        npt = point(cp.row - 1, cp.col, cp)
        if valid_point(maze, npt, 50, 75 ) and not in_fringe_stack(fringe,
            npt) and not visited(closed_matrix, npt):
```

```
            closed_matrix[npt.row][npt.col] = 1
            fringe.append(npt)

        #adding to closed matrix
        closed_matrix[cp.row][cp.col] = 1

    return None, nodes_exp #not solvable
```

In the case where we determine two points in a maze are reachable, DFS is a better choice because BFS will generally use more memory, as it will need to keep multiple paths in memory at the same time, where DFS only needs to keep track of a single path at any given time.



Exercise 3

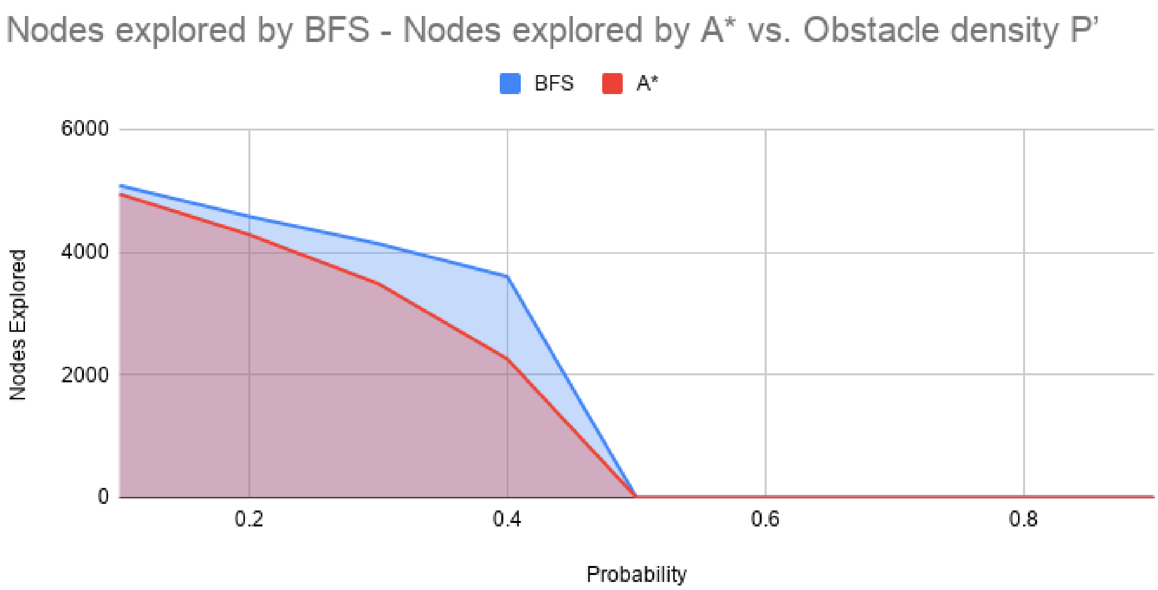
Write BFS and A algorithms(using the euclidean distance metric) that take a maze and determine the shortest path from S to G if one exists. For as large a dimension as your system can handle, generate a plot of the average 'number of nodes explored by BFS - number of nodes explored by A*' vs 'obstacle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

```
from maze_generator import *
from point import *
from maze_functions import *
from queue import Queue
import time


def bfs(start, goal, maze):

    nodes_exp = 0
    fringe = Queue(maxsize=0) #using a queue for fringe
    fringe.put(start)
    closed_matrix = np.zeros((maze.shape[0],maze.shape[0]))

    while not fringe.empty():

        cp = fringe.get()

        nodes_exp += 1 #counting number of nodes explored

        #checking if reached goal
        if is_goal(cp, goal):
            return cp, nodes_exp

        #generating children
        npt = point(cp.row, cp.col + 1, cp)
        if valid_point(maze, npt, 50, 75 ) and not visited(closed_matrix,
             npt):
            closed_matrix[cp.row][cp.col + 1] = 1
            fringe.put(npt)

        npt = point(cp.row, cp.col - 1, cp)
        if valid_point(maze, npt, 50, 75 ) and not visited(closed_matrix,
             npt):
            closed_matrix[cp.row][cp.col - 1] = 1
            fringe.put(npt)


        npt = point(cp.row + 1, cp.col, cp)
        if valid_point(maze, npt , 50, 75 ) and not visited(closed_matrix,
             npt):
            closed_matrix[cp.row + 1][cp.col] = 1
            fringe.put(npt)
```

```
        npt = point(cp.row - 1, cp.col, cp)
        if valid_point(maze, npt, 50, 75 ) and not visited(closed_matrix,
            npt):
            closed_matrix[cp.row - 1][cp.col] = 1
            fringe.put(npt)

    return None, nodes_exp #not solvable
```

```
from maze_generator import *
from point import *
from maze_functions import *
import queue as q
import time


def Astar(start, goal, maze):

    nodes_exp = 0
    fringe = q.PriorityQueue(maxsize=0) #using a priority queue as fringe
    fringe.put(start)
    closed_matrix = np.zeros((maze.shape[0],maze.shape[0]))
    dims = maze.shape[0]

    while not fringe.empty():

        cp = fringe.get()

        nodes_exp += 1

        #checking if reached goal
        if is_goal(cp, goal):
            return cp, nodes_exp

        #generating children
        d_trav = cp.dist_trav + 1 #updating travel distance for each neighbor

        d_left = get_d_left(cp.row, cp.col + 1, dims) #finding estimated
            distance left to goal
        npt = point(cp.row, cp.col + 1, cp, d_trav, d_left, d_trav + d_left)
            #creating new point adding d_trav + d_left for heuristic
        is_in_fringe = in_fringe_P_queue(fringe, npt)
        if valid_point(maze, npt, 50, 75 ) and not is_in_fringe and not
            visited(closed_matrix, npt):
            fringe.put(npt)

        d_left = get_d_left(cp.row, cp.col - 1, dims)
        npt = point(cp.row, cp.col - 1, cp, d_trav, d_left, d_trav + d_left)
        is_in_fringe = in_fringe_P_queue(fringe, npt)
        if valid_point(maze, npt, 50, 75 ) and not is_in_fringe and not
            visited(closed_matrix, npt):
            fringe.put(npt)
```

```
        d_left = get_d_left(cp.row + 1, cp.col, dims)
        npt = point(cp.row + 1, cp.col, cp, d_trav, d_left, d_trav + d_left)
        is_in_fringe = in_fringe_P_queue(fringe, npt)
        if valid_point(maze, npt , 50, 75 ) and not is_in_fringe and not
            visited(closed_matrix, npt):
           fringe.put(npt)

        d_left = get_d_left(cp.row - 1, cp.col, dims)
        npt = point(cp.row - 1, cp.col, cp, d_trav, d_left, d_trav + d_left)
        is_in_fringe = in_fringe_P_queue(fringe, npt)
        if valid_point(maze, npt, 50, 75 ) and not is_in_fringe and not
            visited(closed_matrix, npt):
           fringe.put(npt)

        #adding to closed matrix
        closed_matrix[cp.row][cp.col] = 1


    return None, nodes_exp
```



If there is no path between S and G, the difference between BFS and A* should be 0.

Exercise 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?*

For DFS, the largest dimension possible with p = 0.3 was 275. For BFS, the largest dimension possible with p = 0.3 was 223. For A*, the largest dimension possible with p = 0.3 was 150.

Exercise 5

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

Strategy 3 combines the race-to-the-finish concept of strategy 1 and the slow but safe ideas of strategy 2. Strategy 3 realizes that as long as the fire is not particularly close to the agent there isn't a need to constantly be recalculating a path. Instead the agent will only recalculate if it discovers that one of the cells adjacent to him is on fire. For every time step the agent will look up, down, left, and right and if any of those cells are on fire he will recalculate his path and continue. If none of his neighbors are on fire he will continue on his current path.

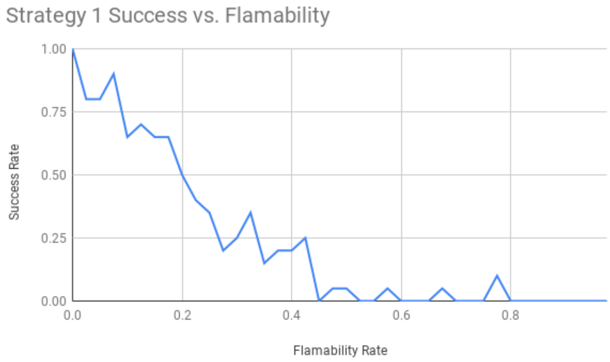
In testing this strategy we found some good things and some bad things. Our strategy did run much faster than strategy 2 with strategy 2 taking about 8 seconds to solve a maze and strategy 3 taking about 3 seconds. However when it came to survivability there was only a marginable increase between strategy 1 and strategy 3.

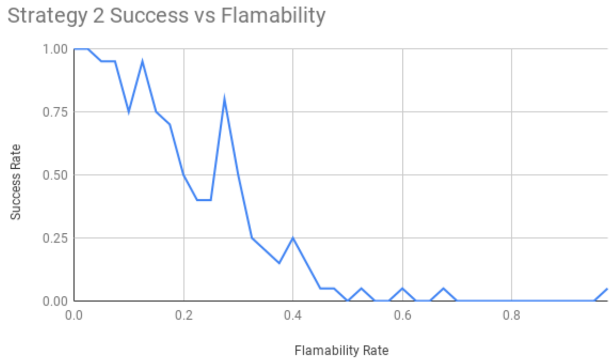
```
if test_strat3:

   print("STRAT 3 TEST:")
   print()

   wall_prob = 0.3
   fire_prob = 0

   while fire_prob < 1:

      st_time = time.perf_counter()
      num_tried = 0
      num_made = 0

      while num_tried < 10:

         maze = maze_generator(maze_dim, wall_prob, True)
         goal_bfs, bfs_exp = bfs(start_point, end_point, maze) #initial bfs
             run
         path_bfs, num_steps_bfs = gen_path(goal_bfs)

         if goal_bfs == None:
            continue
         cp = path_bfs[1]
         count = 0
         burned = False
         new_path = False

         while not is_goal(cp, end_point):

            maze = move_fire(maze, fire_prob, fire_id, wall_id)

            npt1 = point(cp.row, cp.col + 1)
            npt2 = point(cp.row, cp.col - 1)
            npt3 = point(cp.row + 1, cp.col)
            npt4 = point(cp.row - 1, cp.col)
```

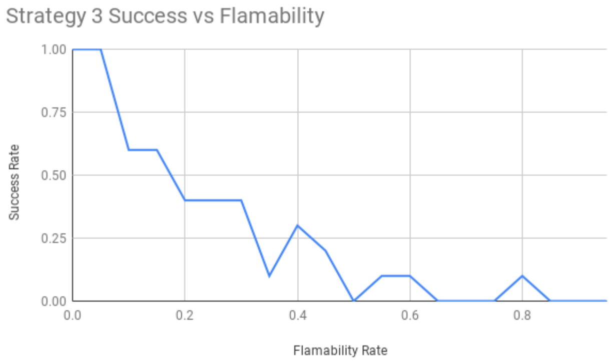
```
        #we check all neighbors of current point and if any are on fire
            we recalculate path
        if is_fire(maze, npt1, fire_id) or is_fire(maze, npt2, fire_id)
            or is_fire(maze, npt3, fire_id) or is_fire(maze, npt4,
            fire_id):

            cp.prev = None
            goal_bfs, bfs_exp = bfs(cp, end_point, maze)
            path_bfs, num_steps_bfs = gen_path(goal_bfs)
            new_path = True

        if goal_bfs == None or is_fire(maze, cp, fire_id): #we burned
            or got trapped
            burned = True
            break
        if new_path:            #we recalculated
            cp = path_bfs[1]
            new_path = False
            count = 2
        else:
            cp = path_bfs[count]
            count += 1

    if not burned:
        num_made += 1
    num_tried += 1

end_time = time.perf_counter()
print("{:.3f},{:.3f},{:.3f}".format(fire_prob, num_made / num_tried,
    end_time - st_time), flush = True)
fire_prob += 0.05
```

Exercise 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?







All three strategies perform the same with a low flammability Q. However, strategy 1 reports the earliest drop in performance, followed by Strategy 2 and lastly Strategy 3. This can be attributed to the properties of each strategy. Strategy 1 does not account for the changing state of the fire, but Strategy 2 does. But Strategy 2 does not account for how the fire is going to look in the future.

Exercise 7

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

Currently strategy 3 only looks at at all cells at a distance of 1 from the agent in an attempt to get out in front of the fire. If more resources were available the agent could look farther and farther from his current position and get even further ahead of the fire. The agent would acquire a much broader view of the fire and where it's going and be able to choose exactly the right moments to recalculate his path without having to sacrifice the time of recalculating at every step. With more resources strategy 3 could maximize both time and success rate by getting a bigger picture of the fire in the maze.

Exercise 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.

If we could only take ten seconds between moves, we could halve the distance towards the goal path at each time step in order to more cost-effectively utilize our strategy, SLEW. First, we STEP + 1 on the maze, advancing our position. Then we LOOK, examining the neighbors and their current fire state. After that, EVALUATE, after calculating the fire probability for each neighbor's potential neighbors, SLEW determines the shortest path relative to these potential spots and WALKS, executing a traversal through the maze to a modified goal state.