CS520 INTRO TO AI ASSIGNMENT 1: EXPLORING SEARCH - THIS MAZE IS ON FIRE

Name: Akash Patel

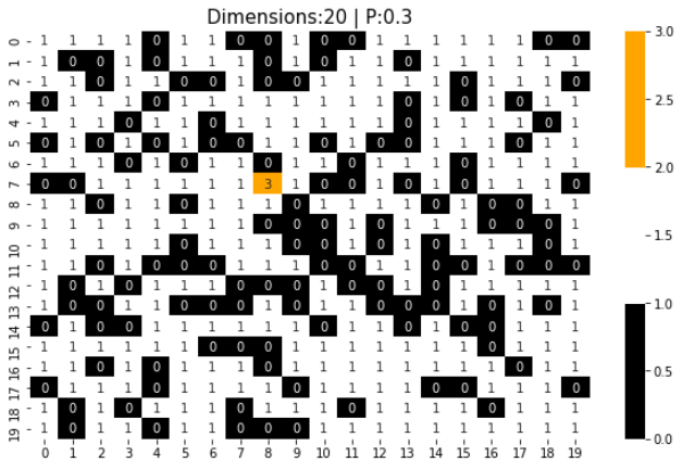
Net id: adp178

First we create maze using np.ones (Creating matrix of given dimension with all values assigned as 1. Now given probability of cell being occupied as “prob”, we use np.random.random() randomly gives a sample(value) if this value is less than prob for each [i,j] of matrix than we assign it as zero. Note: 1 corresponds to open path and 0 as blocked. Now, while randomly choose [i,j] and assign it to 3. 3 corresponds to fire.

```
In [726]: def generate_maze(dimension, prob, show_fig = False):
              maze = np.ones((dimension, dimension))
              for i in range(dimension):
                  for j in range(dimension):
                      if np.random.random() <= prob:
                          maze[i,j] = 0


              maze[0,0] = 1
              maze[dimension-1, dimension-1] = 1
              fire=True
              while(fire):
                  i=randm.randrange(0,dimension-1)
                  j=randm.randrange(0,dimension-1)
                  if(maze[i][j]==0):
                      maze[i][j]=3
                      fire=False
              if show_fig:
                      fig, ax = plt.subplots(figsize=(10,6))
                      plt.grid(True)
                      sns.heatmap(maze, ax=ax,vmin=0,annot= True, cmap=['black','white','Orange'], cbar=True)
                      title = 'Dimensions:{} | P:{}'.format(dimension, prob)
                      ax.set_title(title, fontsize=15)
                      plt.savefig('maze.png')
                      plt.show()
              return maze, prob
```

Maze visualization:

```
In [699]: maze,p = generate_maze(20, 0.3,show_fig = True)
```



Updating Maze as per ‘Flammability rate’ and burning neighbors:

In [836]:
```
def burning_neighbours(row,col,maze):
    if(0<=row<=len(maze)-1 and 0<=col<=len(maze)-1 and maze[row][col]==3):
        return 1
    return 0
```

In [976]:
```
def fire_update(maze,fr):
    temp_maze=maze.copy()
    rd=[-1,1,0,0]
    cd=[0,0,-1,1]
    for i in range(len(maze)-1):
        for j in range(len(maze)-1):

            if(temp_maze[i][j]==0 or temp_maze[i][j]==1):
                total_burning_neighbours=0
                for k in range(4):
                    total_burning_neighbours+=burning_neighbours(i+rd[k],j+cd[k],temp_maze)
                if(total_burning_neighbours>0):
                    prob_fire=pow(1-(1-fr),total_burning_neighbours)
                    if(randm.uniform(0,1)<=prob_fire):
                            maze[i][j] = 3
    return maze
```

Plotting the maze:

In [702]:
```
def plot_search_maze(maze, algo, p):
    if 1 in maze:
        fig, ax = plt.subplots(figsize=(10,6))
        sns.heatmap(maze,annot=True, ax=ax, cmap=["red","blue","black","white","orange"],cbar=True)
        title = '{} | Dimension:{} | p:{}'.format(algo, maze.shape[0], p)
        ax.set_title(title, fontsize=15)
        plt.savefig(algo+'.png')
        plt.show()
    else:
        fig, ax = plt.subplots(figsize=(10,6))
        sns.heatmap(maze,annot=True, ax=ax, cmap=["red","blue","black","orange"], cbar=True)
        title = '{} | Dimension:{} | P:{}'.format(algo, maze.shape[0], p)
        ax.set_title(title, fontsize=15)
        plt.savefig(algo+'.png')
        plt.show()
```

Getting neighbors of the current [i,j] (note: only horizontal or vertical not diagonal)

```
def get_neighbors(maze, curr_node, visited):
    i,j = curr_node

    neighbors = []
    if j+1 < len(maze):
        right = (i,j+1)

        if (maze[right]) and right not in visited and maze[right] != 3 and maze[right] != 0:
            neighbors.append(right)
    if i+1 < len(maze):
        bottom = (i+1,j)

        if (maze[bottom]) and bottom not in visited and maze[bottom] != 3 and maze[bottom] != 0:
            neighbors.append(bottom)

    if i-1 >= 0:
        up = (i-1,j)

        if (maze[up]) and up not in visited and maze[up] != 3 and maze[up] != 0:
            neighbors.append(up)

    if j-1 >= 0:
        left = (i,j-1)

        if (maze[left]) and left not in visited and maze[left] != 3 and maze[left] != 0:
            neighbors.append(left)

    return neighbors
```

STRATEGY 1:

BFS driver code:

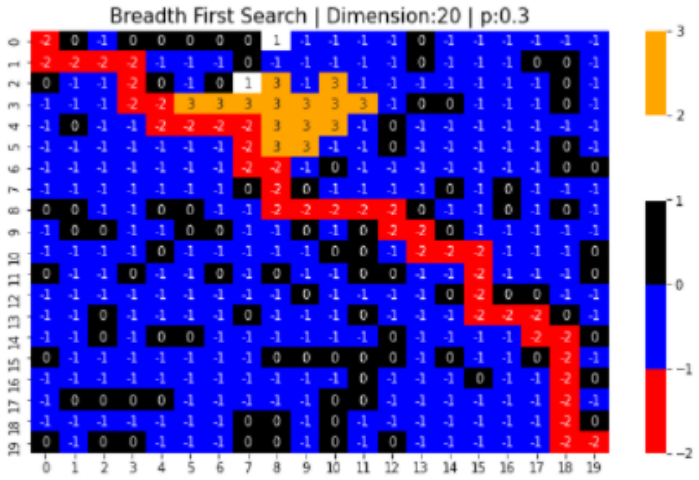
```
In [ ]: def run_bfs(maze,p, show_fig=False):
            start = time.time()
            bfs_maze = maze.copy()

            queue = deque([])
            path = []
            visited = set()
            parent = {}
            curr_node = None
            max_fringe = 1
            source = (0,0)
            goal = (bfs_maze.shape[0]-1, bfs_maze.shape[1]-1)

            queue.append(source)


            while len(queue):

                curr_node = queue.popleft()

                if curr_node not in visited:
                    visited.add(curr_node)
                    bfs_maze[curr_node] = -1

                if curr_node == goal and bfs_maze[curr_node]!=3:


                    time_taken = time.time() - start
                    node = goal
                    while node != source:
                        bfs_maze=fire_update(bfs_maze,0.3)

                        path.insert(0,node)
                        if bfs_maze[node]==3:
                            bfs_maze[node]=6
                        else:
                            bfs_maze[node] = -2


                        node = parent[node]
                    path.insert(0,source)
                    bfs_maze[source] = -2

                    if show_fig:
                        for i in range(len(maze)-1):
                            for j in range(len(maze)-1):
                                if bfs_maze[i][j]==6:
                                    print("Path does not exist")
                                    return 0

                        print('Path exists')
                        plot_search_maze(bfs_maze, 'Breadth First Search', p)

                    return 1,visited, path

                neighbors = get_neighbors(bfs_maze, curr_node, visited)
                for neighbor in neighbors:
                    if neighbor not in parent and bfs_maze[neighbor]!=3 :


                        parent[neighbor] = curr_node
                        queue.append(neighbor)

                if max_fringe < len(queue):
                    max_fringe = len(queue)

            if show_fig:
                print('Path does not exist')
            time_taken = time.time() - start
            return 0, visited, []
```

Here the we initialize maze than call fire_update to spread the fire accordingly. We only call BFS once to get the shortest path from start to goal, and if the path to goal is through a burning cell or is blocked we print path do not exist and return 0.

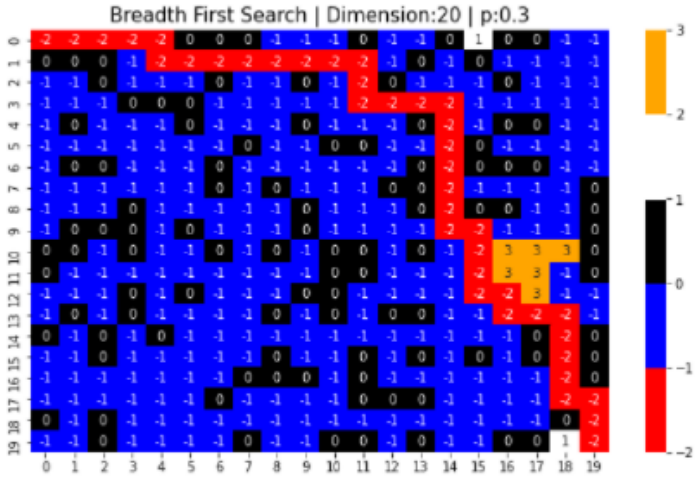
Now, we run 10 trials and results for p= 0.3, q=0.3 and dimension = 20 :

In [1057]:
```
success=0

count=[]
for i in range(1,11):
    maze,p = generate_maze(20, 0.3)

    bfs = run_bfs(maze,p, show_fig= True)

    count.append(bfs[0])
print(count)
for i in range(len(count)):
    if count[i]==1 :
        success+=1
print(success,success/10)
```

```
Path exists
```

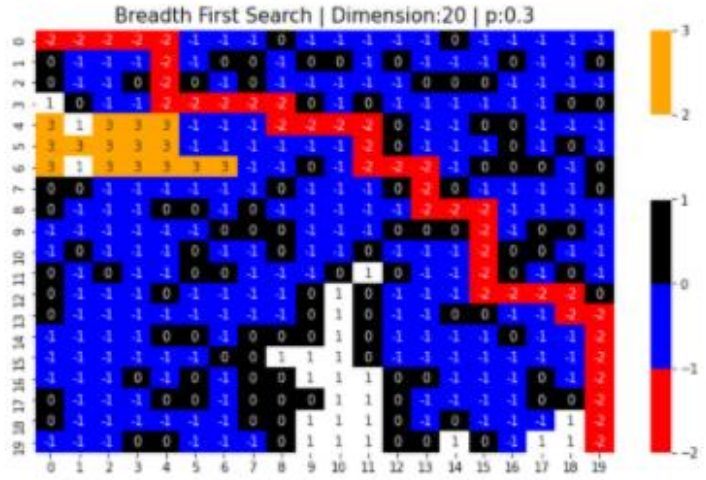



```
Path does not exist
Path exists
```

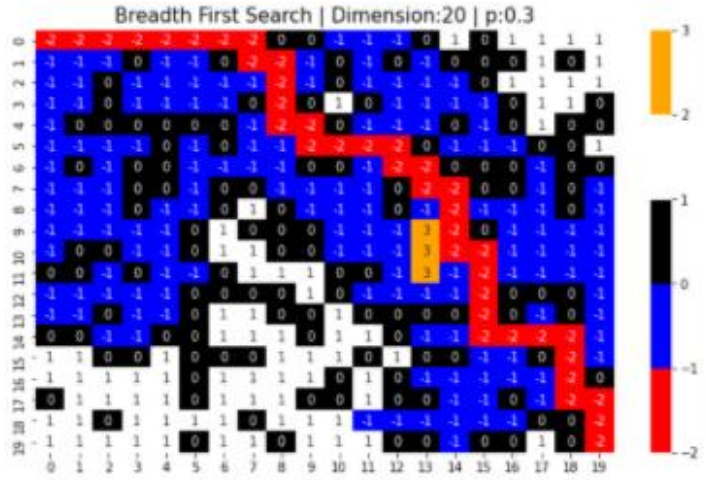



```
Path does not exist
```

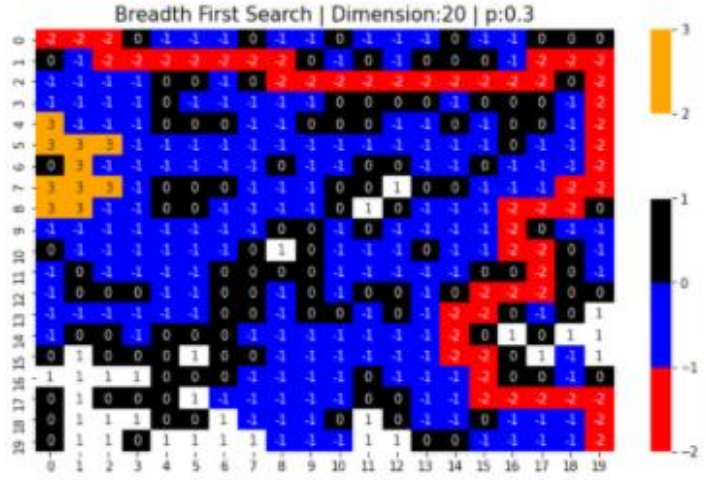
Path does not exist
Path exists



Path does not exist
Path exists



Path does not exist
Path exists

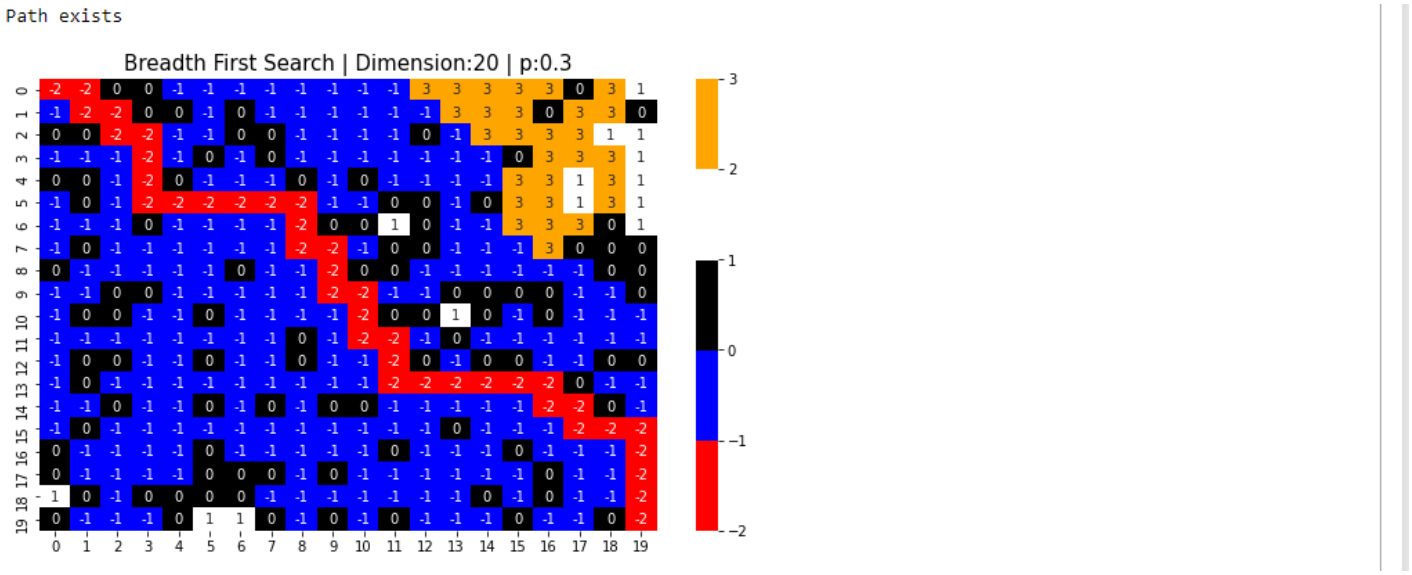


So, we got the shortest path in 5 trials out of 10 for the above given parameters i.e. 0.5 success rate.

Note: -2 corresponds to the path, 0 to the blocked path, -3 to the fire, and -1 are those nodes which were explored.

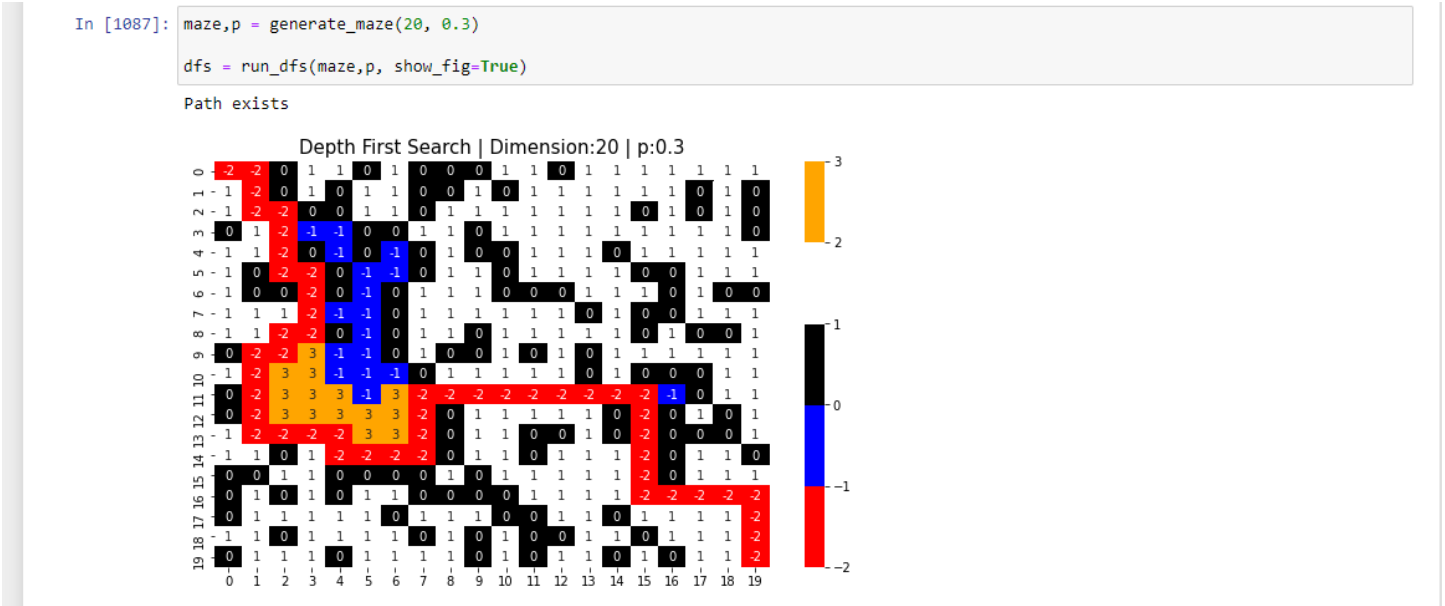
Let's run a single trial :

BFS:



Bonus: Now I had started with dfs first then realized it was asked for the shortest path so had to switch over to bfs as dfs does not guarantee "Optimality". But it was interesting to compare:

DFS:



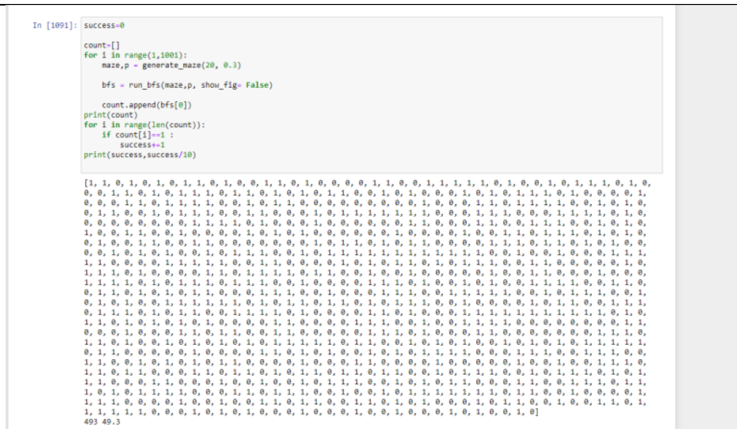
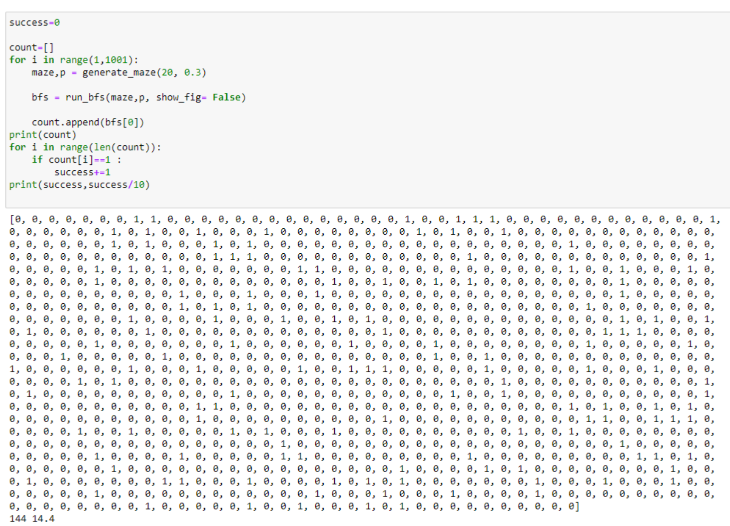
Interestingly, we can see that the nodes explored in DFS are comparatively small in number, now think if this of dimension 10000, we can think of how much computational power we could save. This is also good example to see the difference between DFS and BFS. (**Note**: 1 corresponds to unexplored nodes.)

Now we will be running 1000 runs using BFS to get a plot of `average successes vs flammability q' with p=0.3 and q ranging from 0.1 to 1 and dimension 20x20.

Flammability q	Success rate	Probability of cell being occupied
0.1	success=0 count=[] for i in range(1,1001): maze,p = generate_maze(20, 0.3) bfs = run_bfs(maze,p, show_fig= False) count.append(bfs[0]) print(count) for i in range(len(count)): if count[i]==1 : success+=1 print(success,success/10) [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1] 972 97.2 Success rate: 97.2 %	0.3

0.2	success=0
	
	count=[]
	for i in range(1,1001):
	    maze,p = generate_maze(20, 0.3)
	
	    bfs = run_bfs(maze,p, show_fig= False)
	
	    count.append(bfs[0])
	print(count)
	for i in range(len(count)):
	    if count[i]==1 :
	        success+=1
	print(success,success/10)
	
	[1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
	1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
	0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
	1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,
	0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0,
	1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
	1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0,
	1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0,
	0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
	1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1,
	1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1,
	1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1,
	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1,
	0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1,
	1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1,
	1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
	1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
	1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0,
	1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1,
	1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,
	0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1,
	1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1,
	1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1,
	1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0]
	716 71.6
	
	Success rate: 71.6%
		0.3

0.3	success=0
	
	count=[]
	for i in range(1,1001):
	    maze,p = generate_maze(20, 0.3)
	
	    bfs = run_bfs(maze,p, show_fig= False)
	
	    count.append(bfs[0])
	print(count)
	for i in range(len(count)):
	    if count[i]==1 :
	        success+=1
	print(success,success/10)
	
	[0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,
	1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0,
	1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1,
	0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0,
	0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0,
	1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
	0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,
	1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0,
	0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
	0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1,
	1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
	1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0,
	0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,
	1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0,
	1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
	1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1,
	1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1,
	1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,
	1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1,
	1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1,
	0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1,
	1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,
	1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
	1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]
	523 52.3
	
	Success rate: 52.3%
		0.3

<table>
<tr><th>Flammability q</th><th>Success rate</th><th>Probability of cell being occupied</th></tr>
<tr><td>0.4</td><td> 
Success rate: 49.3</td><td>0.3</td></tr>
<tr><td>0.5</td><td> 
Success rate: 25.9%</td><td>0.3</td></tr>
<tr><td>0.6</td><td> 
Success rate: 14.4%</td><td>0.3</td></tr>
</table>

<table>
<tr><td>Flammability q</td><td>Success rate</td><td>Probability of cell being occupied</td></tr>
<tr><td>0.7</td><td> 
success=0

count=[]
for i in range(1,1001):
    maze,p = generate_maze(20, 0.3)

    bfs = run_bfs(maze,p, show_fig= False)

    count.append(bfs[0])
print(count)
for i in range(len(count)):
    if count[i]==1 :
        success+=1
print(success,success/10)

77 7.7

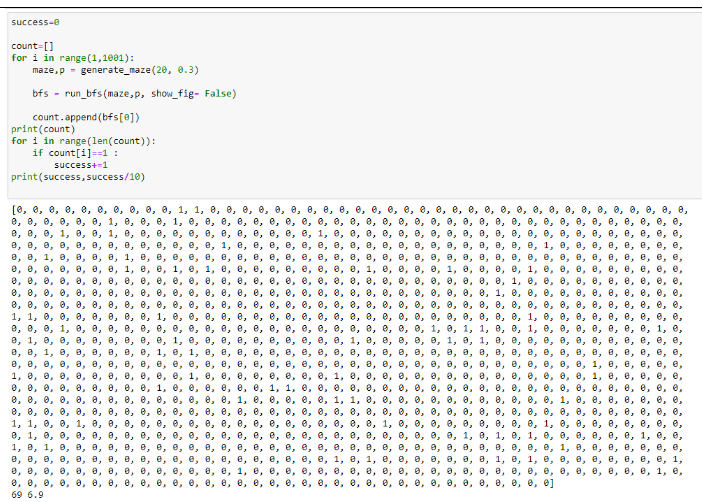
Success rate: 7.7%</td><td>0.3</td></tr>
<tr><td>0.8</td><td> 
success=0

count=[]
for i in range(1,1001):
    maze,p = generate_maze(20, 0.3)

    bfs = run_bfs(maze,p, show_fig= False)

    count.append(bfs[0])
print(count)
for i in range(len(count)):
    if count[i]==1 :
        success+=1
print(success,success/10)

95 9.5

Success rate: 9.5%</td><td>0.3</td></tr>
<tr><td>0.9</td><td> 
success=0

count=[]
for i in range(1,1001):
    maze,p = generate_maze(20, 0.3)

    bfs = run_bfs(maze,p, show_fig= False)

    count.append(bfs[0])
print(count)
for i in range(len(count)):
    if count[i]==1 :
        success+=1
print(success,success/10)

69 6.9

Success rate: 6.9%</td><td>0.3</td></tr>
</table>

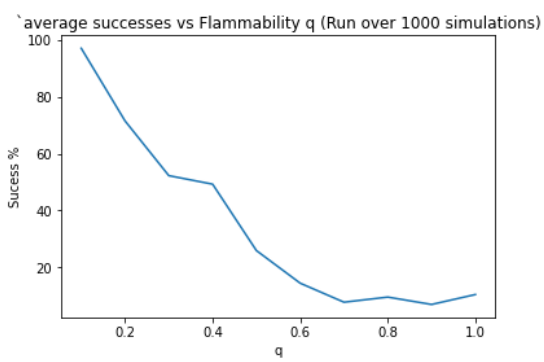
Flammability q	Success rate	Probability of cell being occupied
1.0	Success rate: 10.4%	0.3

```
success=0

count=[]
for i in range(1,1001):
    maze,p = generate_maze(20, 0.39)

    bfs = run_bfs(maze,p, show_fig= False)

    count.append(bfs[0])
print(count)
for i in range(len(count)):
    if count[i]==1 :
        success+=1
print(success,success/10)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0]
104 10.4
```

Average successes vs Flammability q (Run over 1000 runs):

```
In [1285]: df = pd.read_csv('successdata_bfs.csv')
df.head()
x= df['Flammability']
y=df['Success']
plt.plot(x,y)
plt.title('`average successes vs Flammability q (Run over 1000 simulations)')
plt.xlabel('q')
plt.ylabel('Sucess %')
plt.show()
```



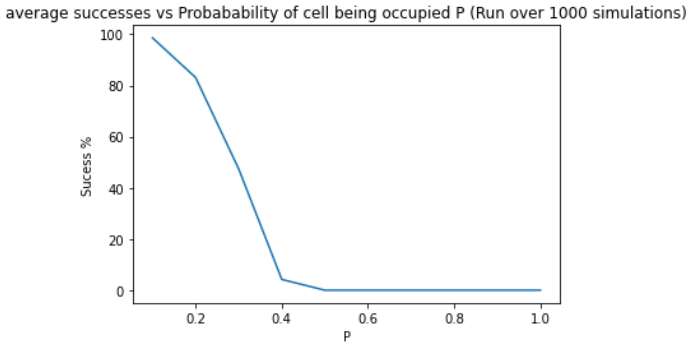
Bonus: Another interesting thing was the relationship between finding success and p (probability of cell being occupied): For 1000 runs

Probability of cell being occupied	Success rate	Flammability q
0.1	98.6	0.3
0.2	83.1	0.3
0.3	47.4	0.3
0.4	4.2	0.3
0.5	0	0.3
0.6	0	0.3
0.7	0	0.3
0.8	0	0.3
0.9	0	0.3

1.0	0	0.3

Average successes vs Probability of cell being occupied P (Run over 1000 simulations)

```
In [1287]: df = pd.read_csv('sucessdata_bfs_p.csv')
           df.head()
           x= df['P']
           y=df['Success']
           plt.plot(x,y)
           plt.title('average successes vs Probabability of cell being occupied P (Run over 1000 simulations)')
           plt.xlabel('P')
           plt.ylabel('Sucess %')
           plt.show()
```



As we can see as **p reaches 0.4 the success rate is nearly zero**, this makes sense as we assuming almost half or more than half cells as occupied as we go beyond 0.4. At first when I thought about it I guessed 0.5 would be the threshold as you have 50% of chance of cell being blocked so half of the cells would be blocked and there would be no path to goal, but as it turns out to be much near to 0.39-0.4, this is something interesting to know.

STRATEGY 2:

In this strategy will be getting the path using BFS and on every node in path we will call run_bfs_strat2 while updating the source with nodes of the path and thus find new path to goal from each point. It will re-adjust the path on position of the fire cells and blocked cells.

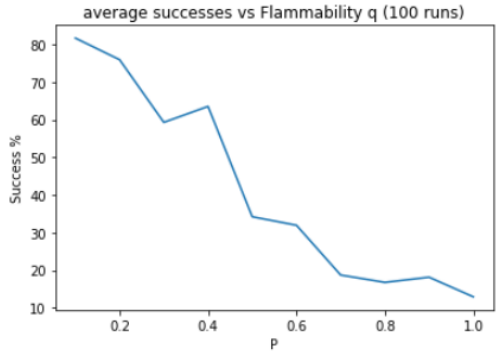
```
In [1510]: #running stratergy 2
           source1=(0,0)
           count=[]
           tot=0
           success=0
           lenfromsource = []
           for i in range(1,11):

               maze,p = generate_maze(20, 0.3)
               bfs = run_bfs(maze,p,show_fig=False)
               new_sources=bfs[2] #nodes in path to goal (will be using as source each time we run bfs_strat2)
               for x in new_sources:
                   source1 = x
                   bfs_maze = fire_update(maze,0.5)#each time we progresss we update the fire
                   bfs1 = run_bfs_strat2(bfs_maze,p,source1,show_fig= False)
                   count.append(bfs1[0])
                   lenfromsource.append(len(source1))
           x=sum(lenfromsource)
           for i in range(len(count)):
               if count[i]==1:
                   success= success+1
           sucess_rate= (success/(x+1))*100
           print(sucess_rate)

           49.8422712933754
```

Now we will be running 100 runs using run_bfs_strat2 to get a plot of `average successes vs flammability q' with p=0.3 and q ranging from 0.1 to 1 and dimension 20x20.

Flammability q	Success rate	Probability of cell being occupied
0.1	81.72	0.3
0.2	75.98	0.3
0.3	59.32	0.3
0.4	63.56	0.3
0.5	34.22	0.3
0.6	31.96	0.3
0.7	18.72	0.3
0.8	16.77	0.3
0.9	18.14	0.3
1	12.93	0.3

Average successes vs Flammability q (100 runs)

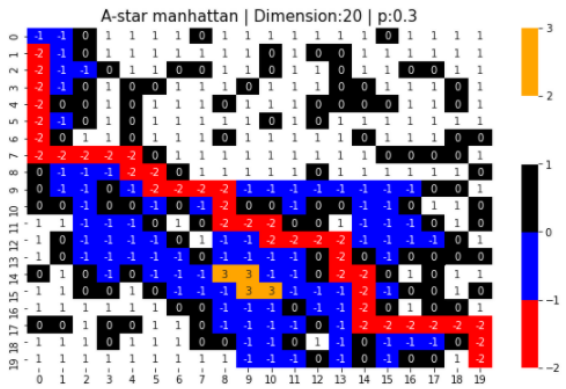
In [1536]:
```
#Plotting sucess rate vs q
df = pd.read_csv('successdata_strat2.csv')
#df.head()
x= df['Flammability']
y=df['Success']
plt.plot(x,y)
plt.title('average successes vs Flammability q (100 runs)')
plt.xlabel('P')
plt.ylabel('Success %')
plt.show()
```

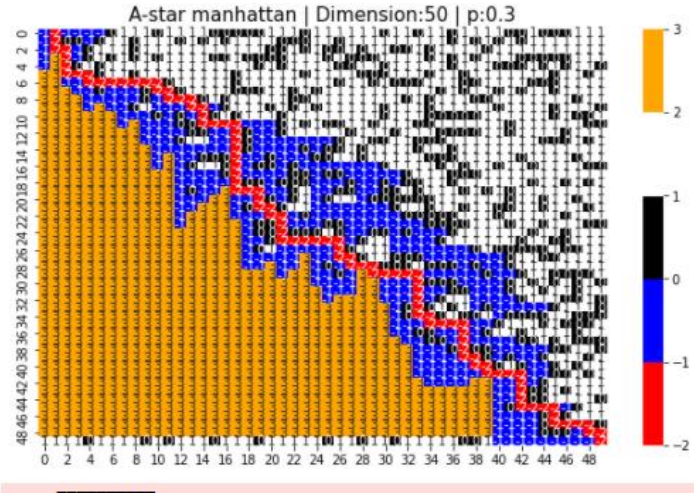


STRATEGY 3:

In this strategy we will be using A* with Manhattan heuristic. As we have only freedom to move in perpendicular directions Manhattan will be more practical to use to calculate cost. Here we take in account the fire rate as well as burning neighbors.

```
In [1609]: maze, prob = generate_maze(20,0.3,show_fig=False)
           A_star = run_A_star(maze, 'manhattan', prob, show_fig=True)

Path exists
```

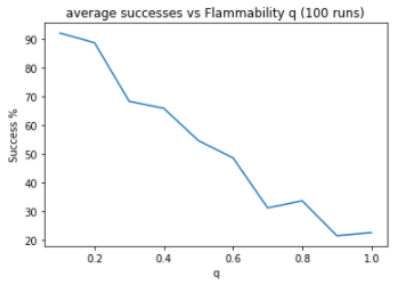




Now we will be running 100 trials to determine the average success rate with p=0.3 and dimension 20x20:

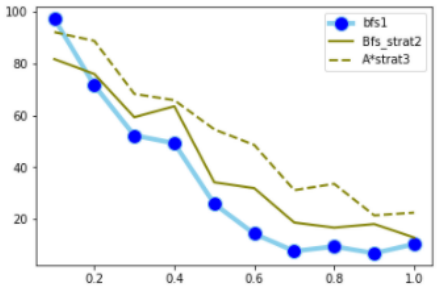
Flammability q	Success rate	Probability of cell being occupied
0.1	92.12	0.3
0.2	88.74	0.3
0.3	68.32	0.3
0.4	65.88	0.3
0.5	54.63	0.3
0.6	48.62	0.3
0.7	31.21	0.3
0.8	33.63	0.3
0.9	21.49	0.3
1	22.56	0.3

```
In [1611]: #Plotting sucess rate vs q
df = pd.read_csv('successdata_strat3.csv')
#df.head()
x= df['Flammability']
y=df['Success']
plt.plot(x,y)
plt.title('average successes vs Flammability q (100 runs)')
plt.xlabel('q')
plt.ylabel('Success %')
plt.show()
```



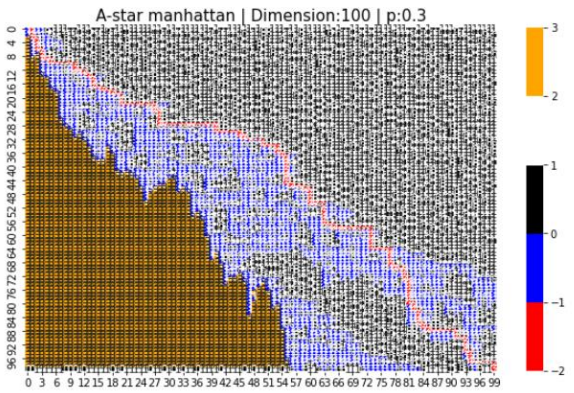
Now comparing the three:

```
Out[1624]: <matplotlib.legend.Legend at 0x16f95f25b08>
```



Observation: we can see that after q=0.2, the strategy 2 has higher success rate than strategy 1 and strategy 3 has higher rate than strategy 2. Note that this was for small dimension (20) and strat1 was ran 1000 times and strat2 and strat3 100 times. We can get better convergence to true results if we increase the dimension to near 500 and run the trial more than 10000 times because that will directly affect the generation of mazes and these mazes will in turn incorporate the probability factor associated with q and p. Just to get an idea how does a maze look when u just increase the dimension to 100.

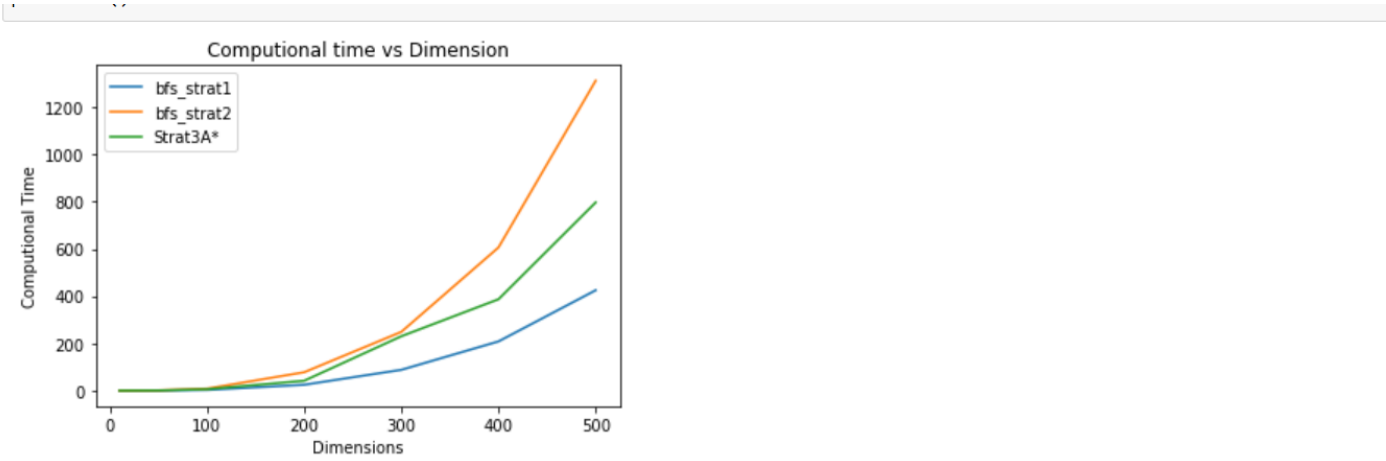
```
In [1672]: maze, prob = generate_maze(100,0.3,show_fig=False)
A_star = run_A_star(maze, 'manhattan', prob, show_fig=True)
Path exists
```



DIMENSION VS COMPUTATIONAL TIME ANALYSIS

Dimension (p:0.3,q:0.3)	COMPUTATIONAL TIME (in seconds)		
	Bfs_strat1	Bfs_strat2	Strat3A*

10	**0.010970830917358398**	**0.013130664825439453**	**0.027924537658691406**
50	**0.4029207229614258**	**1.170900821685791**	**0.7530176639556885**
100	**3.0341875553131104**	**8.28887391090393**	**6.11536431312561**
200	**24.704742908477783**	**77.7760374546051**	**41.683945417404175**
300	**88.01867985725403**	**248.70536851882935**	**230.10630702972412**
400	**208.2877938747406**	**606.1554141044617**	**386.77769231796265**
500	**424.88025069236755**	**1312.0530846118927**	**796.7528831958771**



It is important to note that while running the strategies I keep the show_fig False if I had kept it true it would have taken hours(or days if I went d>2500 I guess) instead of minutes. The best computational time performance is given by bfs_strat1 and that makes sense because we are doing least amount of work compared top the other two. Now it is important to note that as we keep increasing dimension time taken by bfs2_strat 2 is quiet high compared to the other two. Algorithmically this is because we are running bfs at each node in the path to compute the shortest path, this makes strategy much more intensive. Strat3A* is taking more time than bfs_strat1 as we need to compute cost for the nodes. Also as seen from the graph strat2 is the most time exhaustive.

Finally, I found this assignment to be intensive (may be because I was not part of the group) but a very good learning opportunity. Also, would like to seniors and other classmates for the engaging discussions that helped a lot. Please let me know if there is something that can be done better or any new ideas that can be implemented. Thank you.