final

March 20, 2021

1 Maze on Fire

Intro to AI Project 1

Reagan McFarland (rpm141), Alay Shah (acs286), Toshanraju Vysyaraju (tv135)

Imports: - The only non standard import here is trange from tqdm, but all uses of `trange` can be replaced with `range` with no change in results - just no neat progress bar :)

[ ]:
```
# imports
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap
from copy import deepcopy
import random
import numpy as np
from tqdm.notebook import trange
from tqdm.notebook import tqdm
from queue import PriorityQueue, Queue
from pprint import pprint
import math
```

1.1 Problem 1

1.1.1 Generating a maze

When thinking about how we were going to generate the maze, we first wanted to figure out a good way to render the maze in `jupyter notebooks`. This led us to color maps in `matplotlib` using a custom color map, allowing us to mark nodes certain colors based on the value. This resulted in us going with a 2D list of boolean values, where: - `False` = open cell - `True` = occupied cell

We also wanted to make it as easy as possible to reuse, so we have optional params `p` and `dim`, representing the obstacle density and dimension respecitively. We also have the optional params `start` and `goal` as a tuple, that we can use to force those spaces to be empty. These optional params get re-used in almost every function. This is all done in the function `gen_maze`.

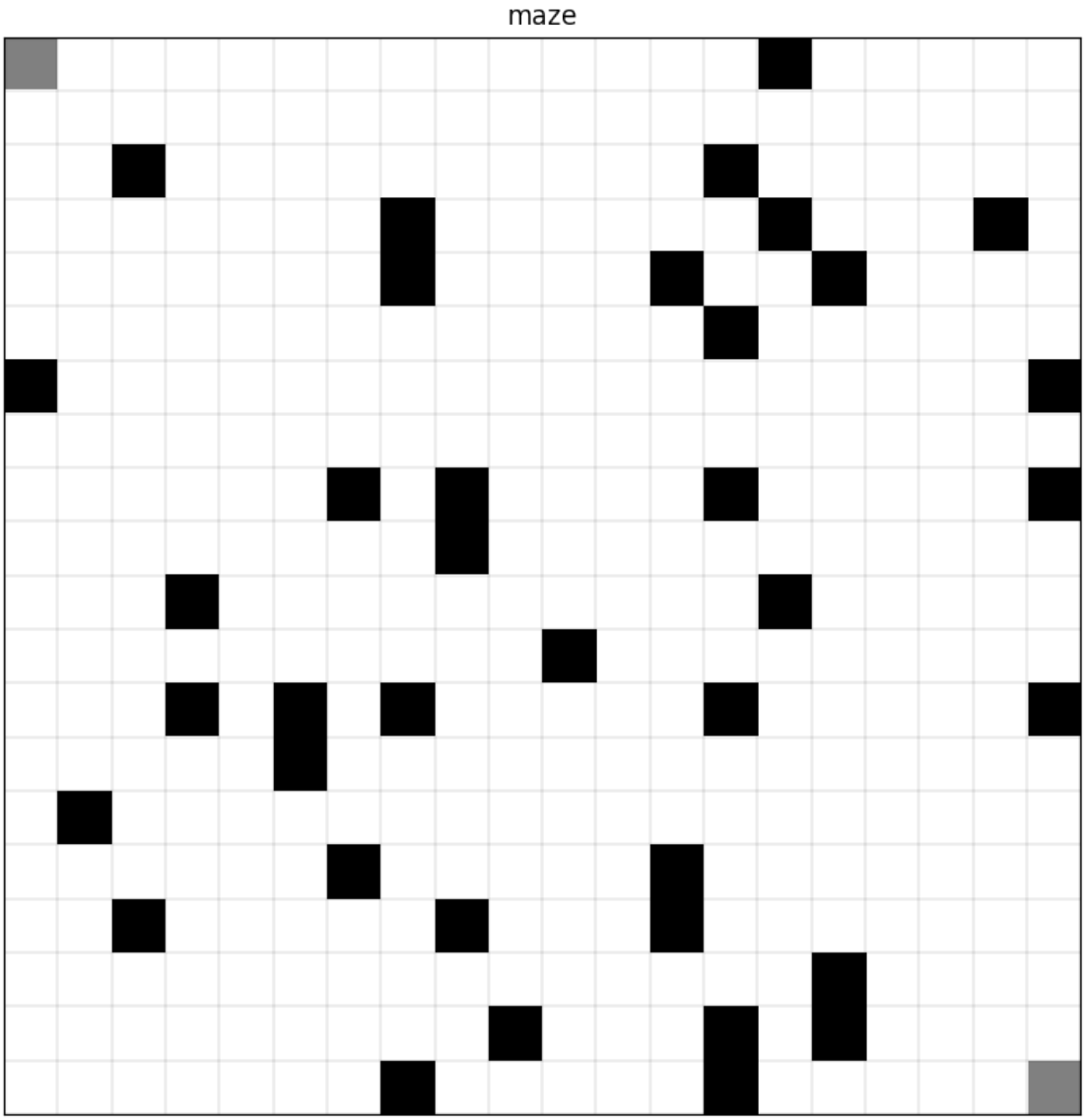
1.1.2 Rendering a maze

We also wanted a method to show a color map plot of our maze, and settled on the following function after reading the `matplotlib` docs.

Notice that we can pass in the index as a tuple for both the start and end if needed. This is because we render these as gray instead of white / black making it easier to see the start and goal of the maze. These are optional params though of course. This is done in the function `render_maze`.

Putting these together, we can generate an example maze with `p=0.3` and `dim=20`

[ ]:
```
example_maze = gen_maze()
# example_maze2 = gen_maze(0.5, 50) # we could also do this if we wanted to␣
 ↪specify the object density and dimensions explicitly
render_maze(example_maze)
```



1.2 Problem 2

1.2.1 Our DFS Algorithm

DFS is a very rudimentary search algorithm, with the only real decision to be made when implementing is whether or not you want to a recursive or iterative approach. We decided to go with a iterative approach just because its easier to transform into BFS later. We also have optional params here for our start and end cell indexes because our start and end goals are pre-determined for most of our test cases, but can be changed if needed with little alteration. We also have a traceNodes optional parameter which will highlight then nodes visited whenever we pass the maze to `render_maze()`. All this is implemented in the function `DFSUninformed`.

1.2.2 Why is DFS a better choice than BFS here?

DFS is a better choice than BFS here because we do not need to find an optimal path. This allows us to save on the space complexity as our fringe is exponentially smaller as we proved in class. In addition, we know the goal node to be the furthest point from the start, i.e. the deepest part of the graph. DFS benefits greatly because it goes further down one path rather than exploring all neighbors, like BFS. Therefore as DFS searches for the 'deepest' node, it is a much better option in this scenario.

1.2.3 Obstacle Density `p` vs. Probabilty that `S` can be reached from `G`

Our machine could handle dimension of 100 with 100 samples per object density `p` we test. We are testing all `p` between `0` and `1`, with a step of `0.1` for each iteration.

1.3 Problem 3

1.3.1 Our BFS Algorithm

BFS implements a queue instead of a stack in order to visit all neighbors before visiting children. This is done in our function `BFSUninformed` which returns if the goal is reached, the number of visited cells, the length of the path to the goal, and the actual path it took.

Before we begin, we first wanted to define a function `trace_path` that would trace the optimal path so we could visually see it on our graph. It will also return the length of the optimal path and the actual path it self (in a stack).

1.3.2 Our A* Algorithm

Before we implement our `A*` algorithm, we need to define a function that can give us the euclidean distance between a cell and the goal cell. This is done in the function `euclid`.

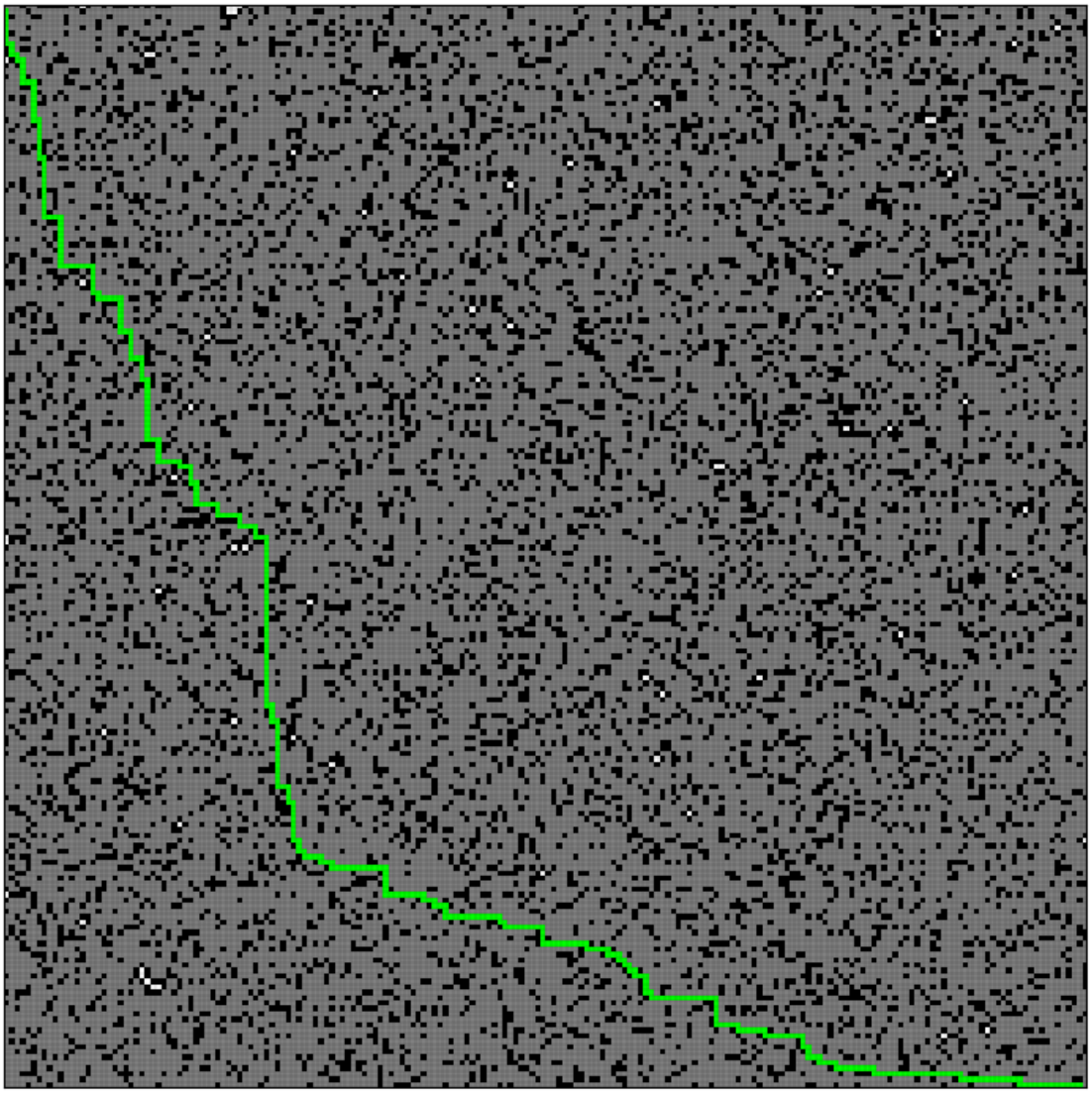
With the heuristic defined, we can now write the A* algorithm in the function `AStar` using almost the same functionality as BFS. However, we are using a `PriorityQueue` instead of a normal queue. The priority queue orders cells with a lower cost based on the heuristic and actual cost from low to high.

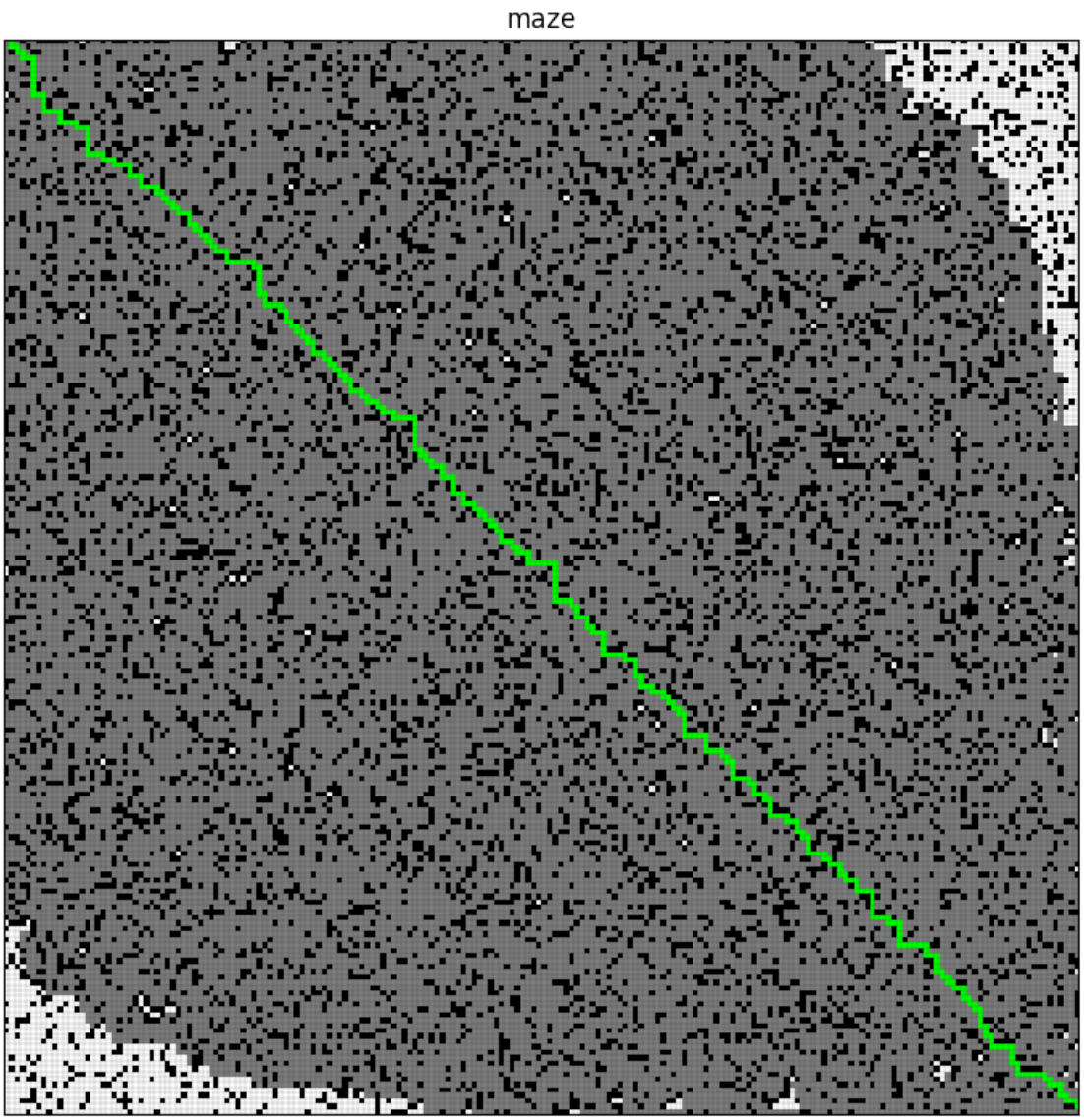
We can view a random example of how the number of nodes A* visits vs. BFS differs. In the below mazes, the grey represents cells that were visited and the green represents the optimal path returned. The white cells are the cells which are not visited. It is important to note that there are multiple optimal paths and so the BFS path may not be identical to the A* path but they both have identical length.

[ ]:
```
maze_bfs = gen_maze(0.2, 200)
maze_astar = deepcopy(maze_bfs)
goal=(len(maze_bfs)-1, len(maze_bfs)-1)
bfs = BFSUninformed(maze_bfs, goal=goal, traceNodes=True)
astar = AStar(maze_astar,goal=goal, traceNodes=True)
print("BFS Nodes Visited = " + str(bfs[0:3]))
render_maze(maze=maze_bfs, goal=goal, traceNodes=True)
print("A* Nodes Visited = " + str(astar[0:3]))
render_maze(maze=maze_astar, goal=goal, traceNodes=True)
```

```
BFS Nodes Visited = (True, 31817, 398)
A* Nodes Visited = (True, 30244, 398)
```

maze





1.3.3 Number of nodes explored by BFS - number of nodes explored by A* vs. obstacle density `p`

In order to graph this, we are going to use the same assumption for the last graph about sample sizes, steps, etc. On top of this, we are going to create a function that for every single sample, will do the following: 1. Generate a new maze 2. Run BFS and record the number of nodes explored 3. Run A* and record the number of nodes explored

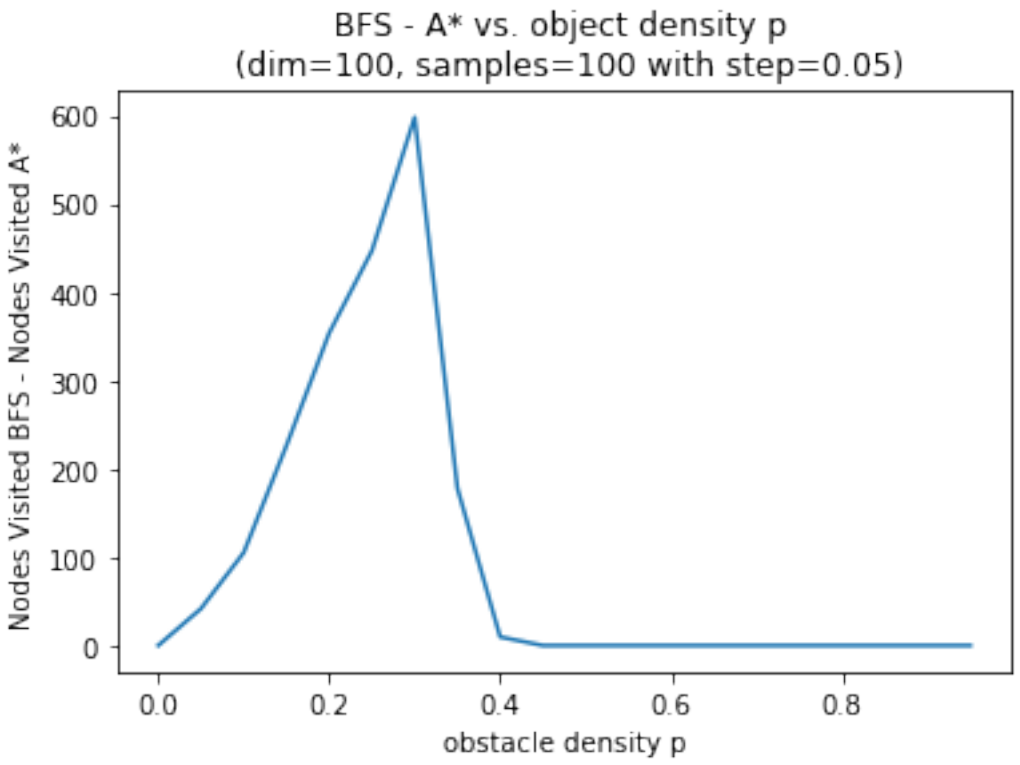
Then, at the end we will average these out across the steps and graph it using matplotlib. First, let's create that function `diff_AStar_BFS` that will do all the 3 steps above.

Now, let's write some code to generate to generate the data for the graph and then also render it, just like before.

[ ]:
```
# Settings
SAMPLE_COUNT = 100
STEP = 0.05
DIMENSION = 100

# Code to generate graph
densities = np.arange(0, 1, STEP).tolist()
successes = dict()
density_count = len(densities)
with tqdm(total=density_count * SAMPLE_COUNT) as pbar:
    for i in range(len(densities)):
        p = densities[i]
        if p not in successes:
            successes[p] = 0
        for j in range(SAMPLE_COUNT):
            successes[p] += diff_AStar_BFS(p, DIMENSION)
            pbar.update(1)

x_axis = densities
y_axis = [successes[x] / SAMPLE_COUNT for x in densities]

plt.xlabel("obstacle density p")
plt.ylabel("Nodes Visited BFS - Nodes Visited A*")
plt.title("BFS - A* vs. object density p \n (dim=" + str(DIMENSION) + ",␣
 ↪samples=" + str(SAMPLE_COUNT) + " with step=" + str(STEP) + ")")
plt.plot(x_axis, y_axis)
```

```
HBox(children=(FloatProgress(value=0.0, max=2000.0), HTML(value='')))
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7fd6b1f15128>]
```



1.3.4 If there is no path from S to G, what should this difference be?

If there is no path, then the difference will be 0. This is because both algorithms will have to check all the same nodes before it can 100% be sure that there is no path. Both algorithms are using a queue, its just that A* will probably get there faster because its using a priority queue with a heuristic. The algorithm cannot know that the path is blocked and the heurestic is rendered useless, so in both cases, it will explore each possibility in hopes of finding the goal node. Therefore, the difference in the number of nodes traversed for each algorithm will be the same when there is no path.

1.4 Problem 4

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

Through trial and error, we found that the largest maze with density `p=0.3` that we can will solve is around 4350x4350

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

Through trial and error, we found that the largest maze with density `p=0.3` that we can will solve is around 4500x4500

1.4.3 What's the largest dimension you can solve using A* at `p=0.3` in less than a minute?

Through trial and error, we found that the largest maze with density `p=0.3` that we can solve is around 15000x15000

1.4.4 Consider, as you solve these three problems, simple diagnostic criteria to make sure you areon track. The path returned by DFS should never be shorter than the path returned by BFS.The path returned by A* should not be shorter than the path returned by BFS. How big can andshould your fringe be at any point during these algorithms?

In the maze, each node has at most 4 neighbors (up, down, left, right). In class we derived the space complexity of the fringes. Here, we will treat A* and BFS as the same, since we are dealing with worst case (i.e. $h(n)$ is the same for all nodes). Let n be the size of the maze. Then we arrive at a space complexity of $O(4n)$ for DFS. However, in the case of BFS and A*, the space complexity of our fringe will be $O(4^n)$. At runtime, the A* algorithm might have a fringe smaller than $O(4^n)$ as it only explores the nodes that bring it closer to the goal node, but ultimately it is also bounded by a space complexity of $O(4^n)$.

2 Part 2 Maze on Fire

Looking at non-static mazes, there is a fire that is acitvely burning down and we need to get out before running into the fire. Solving for the current state may not work for future states of the maze.

2.1 Generating a maze

We decided to re-use our `gen_maze(dim, p)` function to create the maze and then choose a random cell to be on fire. To ensure that the goal and start nodes are not the starting points of the fire, we used a function `randomFirestart(dim)` to properly pick the starting point of the fire. We then use the method `gen_fireMaze` to generate a maze with a random fire start cell.

2.1.1 Expanding the fire

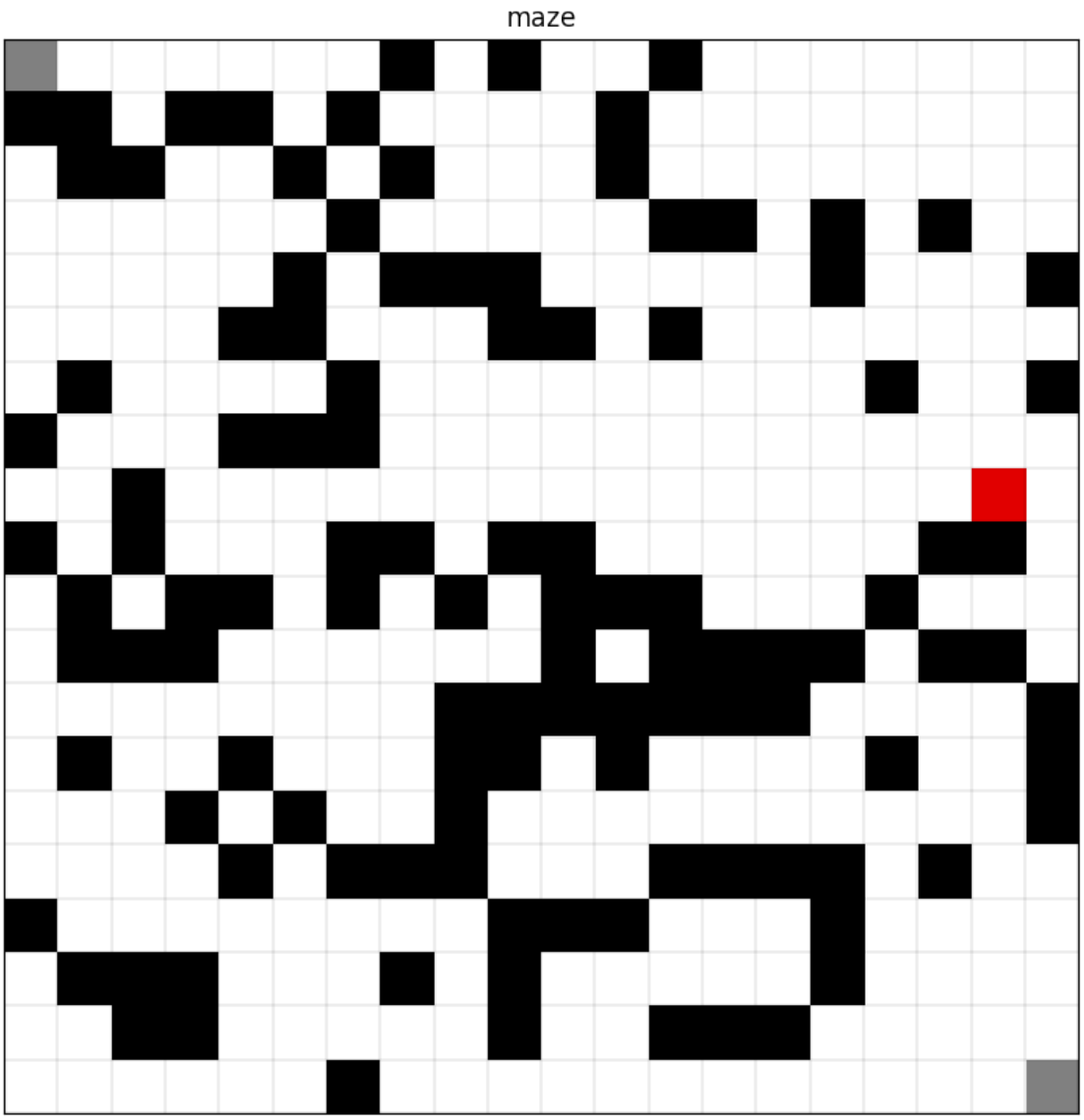
We create a matrix that houses the current state of the fire, and based on that will determine which cells will be on fire in the method `expandFireOneStep`, using the stated parameters:

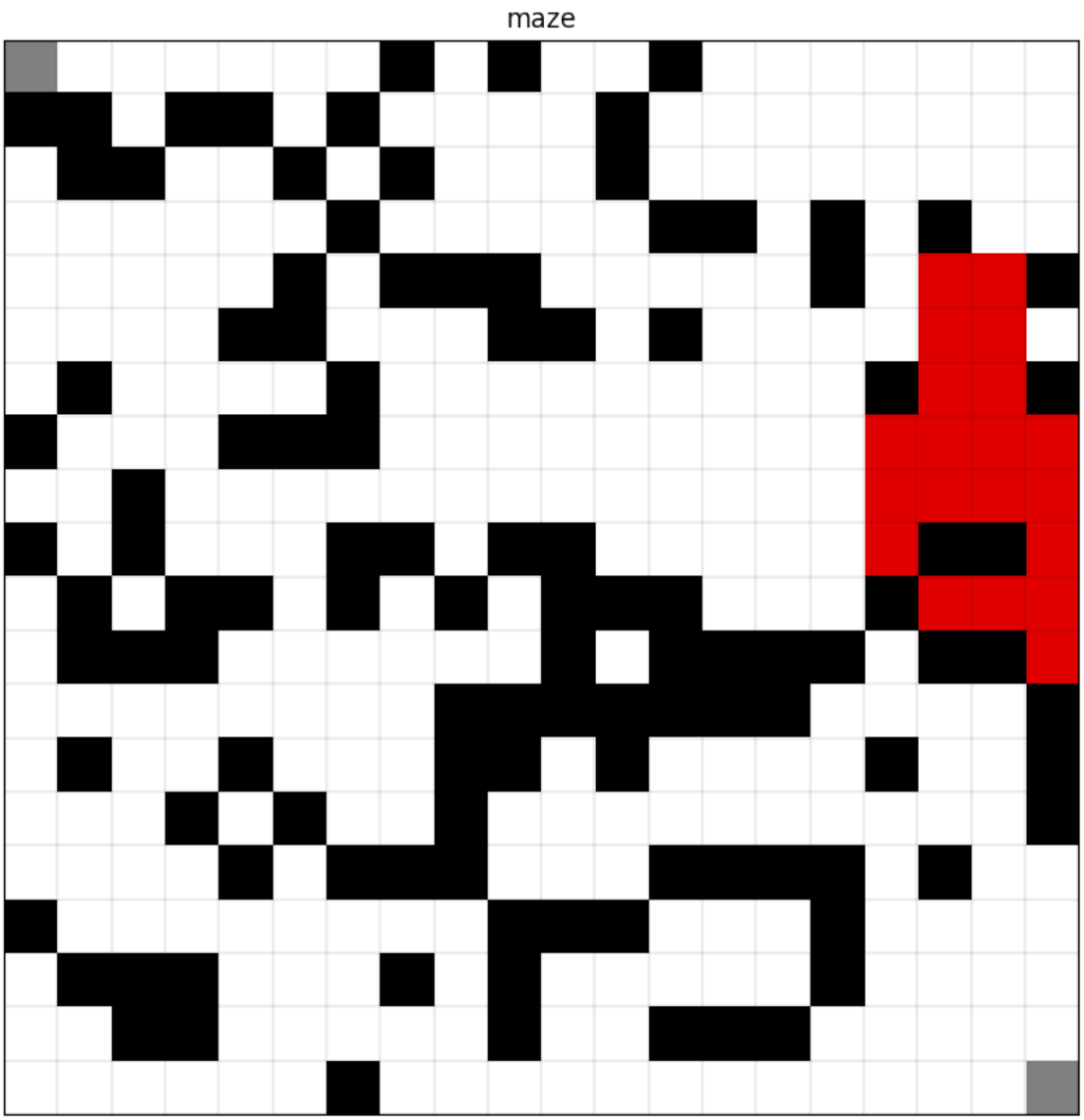
- If a free cell has no burning neighbors, it will still be free in the next time step.
- If a cell is on fire, it will still be on fire in the next time step.
- A blocked cell cannot catch on fire.
- If a free cell has k burning neighbors, it will be on fire in the next time step with probability $1(1q)^k$

We can illustrate this function by rendering maze with `dim=20, p=0.3` ,and`q=0.3`, and having the fire expand 10 steps.

[ ]:
```
maze,_ = gen_fireMaze(dim=20, p=0.3)
render_maze(maze,goal=(19,19),fire=True)
for i in range(10):
    maze = expandFireOneStep(maze, 0.3)
```

```
render_maze(maze,goal=(19,19),fire=True)
```





2.2 Strategy 1, No Strategy

We will use A* to calculate the shortest path to the goal and follow as if there was no fire in the method `strat_one`. This strategy blindly follows the path that A* returns and does not account for the current or future states of the fire.

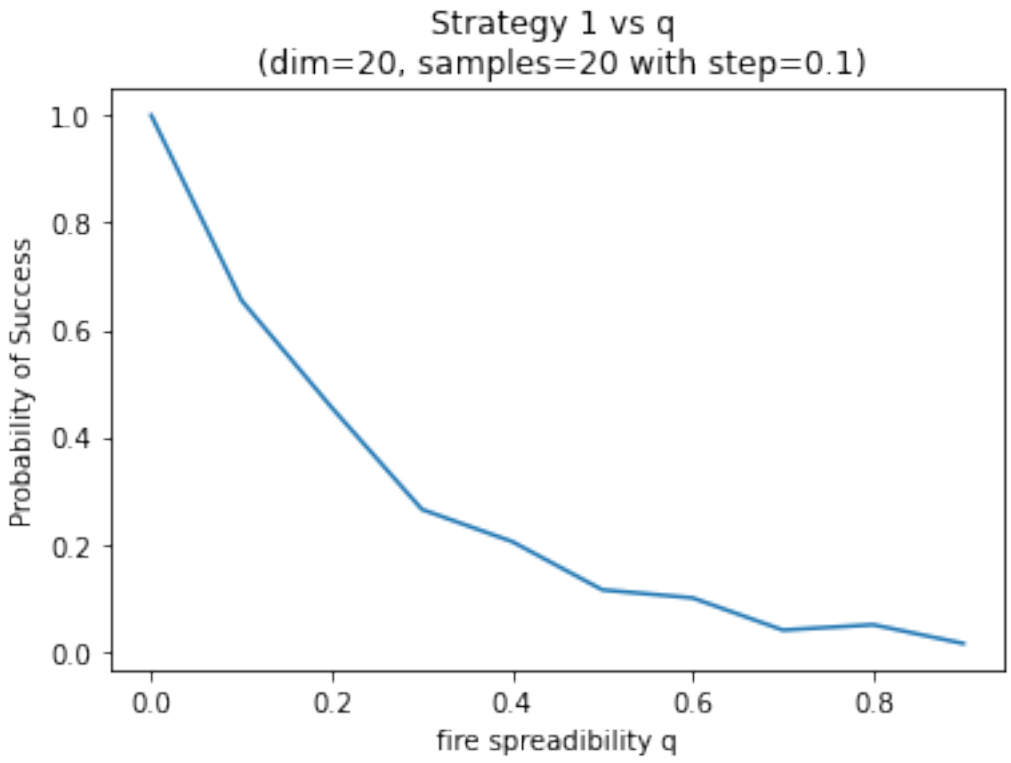
With strategy 1 now well defined, let us create a method `test_strat1` to test it easily for a given dimension and fire spreading rate. For each test, we will generate the fire start position randomly 10 times and return the number of successes encountered.

Now with our testing method `test_strat1` defined above, we can average out probability of success over a defined interval and graph the results. In the graph shown below, we ran Strategy 1 with `dimension 20`, `sample count 20`, and increased `q` by `0.1`.

[ ]:
```
# Settings
SAMPLE_COUNT = 20
STEP = 0.1
DIMENSION = 20

successes = dict()

# Code to generate graph
densities = np.arange(0, 1, STEP).tolist()
density_count = len(densities)
with tqdm(total=density_count * SAMPLE_COUNT) as pbar:
    for i in range(len(densities)):
        q = densities[i]
        if q not in successes:
            successes[q] = 0
        for j in range(SAMPLE_COUNT):
            successes[q] += test_strat1(dim=DIMENSION, q=q)
            pbar.update(1)

x_axis = densities
strat_1_y_axis = [successes[x] / (SAMPLE_COUNT * 10) for x in densities]

plt.xlabel("fire spreadibility q")
plt.ylabel("Probability of Success")
plt.title("Strategy 1 vs q \n (dim=" + str(DIMENSION) + ", samples=" +␣
 ↪str(SAMPLE_COUNT) + " with step=" + str(STEP) + ")")
plt.plot(x_axis, strat_1_y_axis)
```

```
HBox(children=(FloatProgress(value=0.0, max=200.0), HTML(value='')))
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7fee2e2ee518>]
```



2.3 Strategy 2, Recompute Path at Every Step

Strategy 2 is very similar to Strategy 1 but we recalculate our A* path at every step making sure to update our path to take into account the changes in our fire. This is done in the method `strat_two`.

With strategy 2 now well defined, let us create a method `test_strat2` to test it easily for a given dimension and fire spreading rate. For each test, we will generate the fire start position randomly 10 times and return the number of successes encountered.

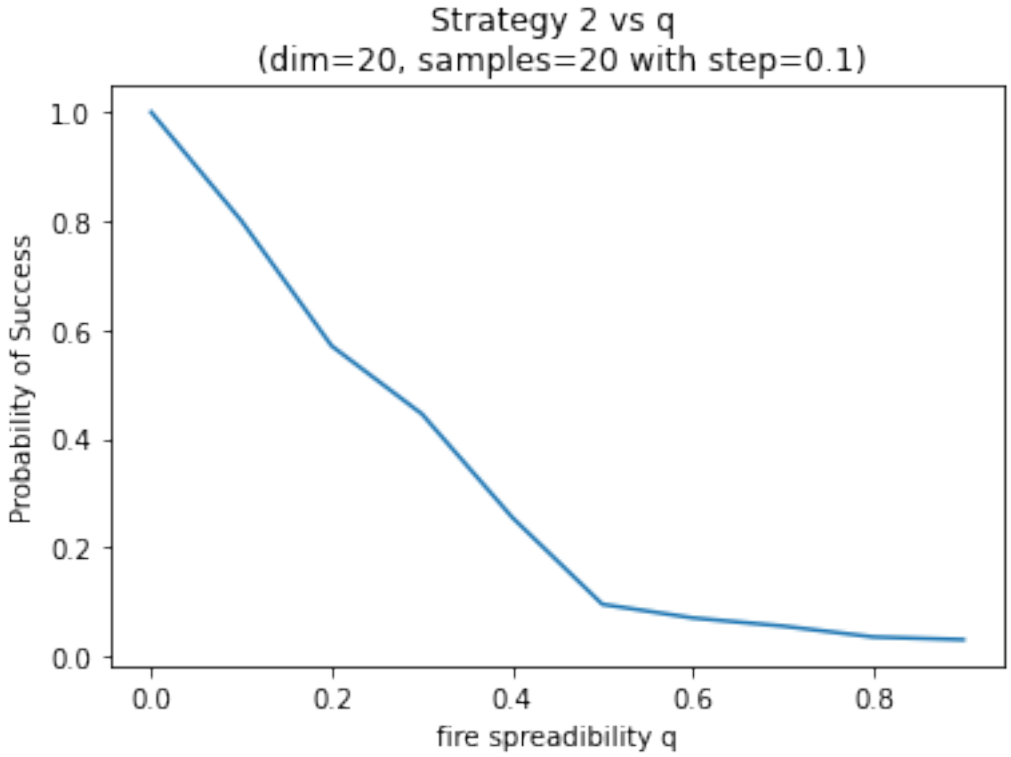
Now with our testing method `test_strat2` defined, we can average out probability of success over a defined interval and graph the results. In the graph shown below, we ran Strategy 2 with `dimension 20`, `sample count 20`, and increased `q` by `0.1`.

[ ]:
```
# Settings
SAMPLE_COUNT = 20
STEP = 0.1
DIMENSION = 20

successes = dict()

# Code to generate graph
densities = np.arange(0, 1, STEP).tolist()
density_count = len(densities)
```

```
with tqdm(total=density_count * SAMPLE_COUNT) as pbar:
    for i in range(len(densities)):
        q = densities[i]
        if q not in successes:
            successes[q] = 0
        for j in range(SAMPLE_COUNT):
            successes[q] += test_strat2(dim=DIMENSION, q=q)
            pbar.update(1)

x_axis = densities
strat_2_y_axis = [successes[x] / (SAMPLE_COUNT * 10) for x in densities]

plt.xlabel("fire spreadibility q")
plt.ylabel("Probability of Success")
plt.title("Strategy 2 vs q \n (dim=" + str(DIMENSION) + ", samples=" +␣
 ↪str(SAMPLE_COUNT) + " with step=" + str(STEP) + ")")
plt.plot(x_axis, strat_2_y_axis)
```

```
HBox(children=(FloatProgress(value=0.0, max=200.0), HTML(value='')))
```

[ ]: [<matplotlib.lines.Line2D at 0x7fd6b350a5f8>]



2.4 Problem 5 - Strategy 3, Future Risk Adjusted Path (FRAP)

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

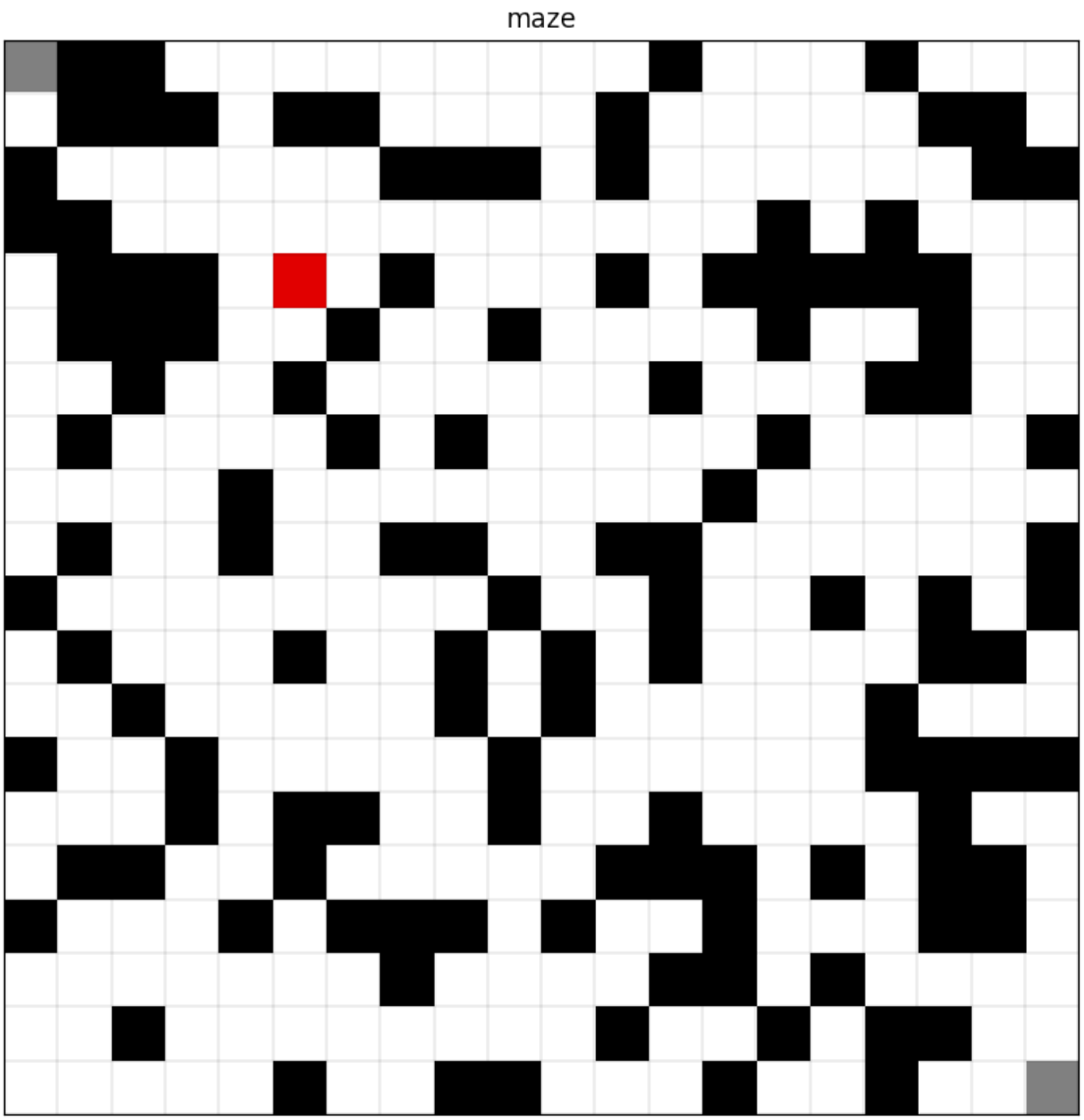
Strategy 3 is our way to solve this maze on fire problem. The problem with Strategy 2 is that it fails to take into account the future state of the maze, resulting in sub-optimal results. Unlike Strategy 1 and Strategy 2, our Strategy 3 takes into account not only the current state of the fire but also potential future states of the fire. To do this, we generate a fire map that takes in the current state of the maze, including what is on fire currently. From this, assuming a worst case `q = 1.0` fire spread probability, we calculate at what step each cell will catch on fire and is therefore a representation of the future of the fire. This is then used to create a weighting for each cell. This weighting is based on the cell's proximity to the goal as well as at which step it will be on fire in the worst case (based on the fire map) and at which step the cell will be visited by our agent. These weights are used as the heuristics for A*. More specifically, the cost of the cell in the fringe is equal to the manhattan distance to the goal + the cost to reach the goal - the cell's value in the fire map.

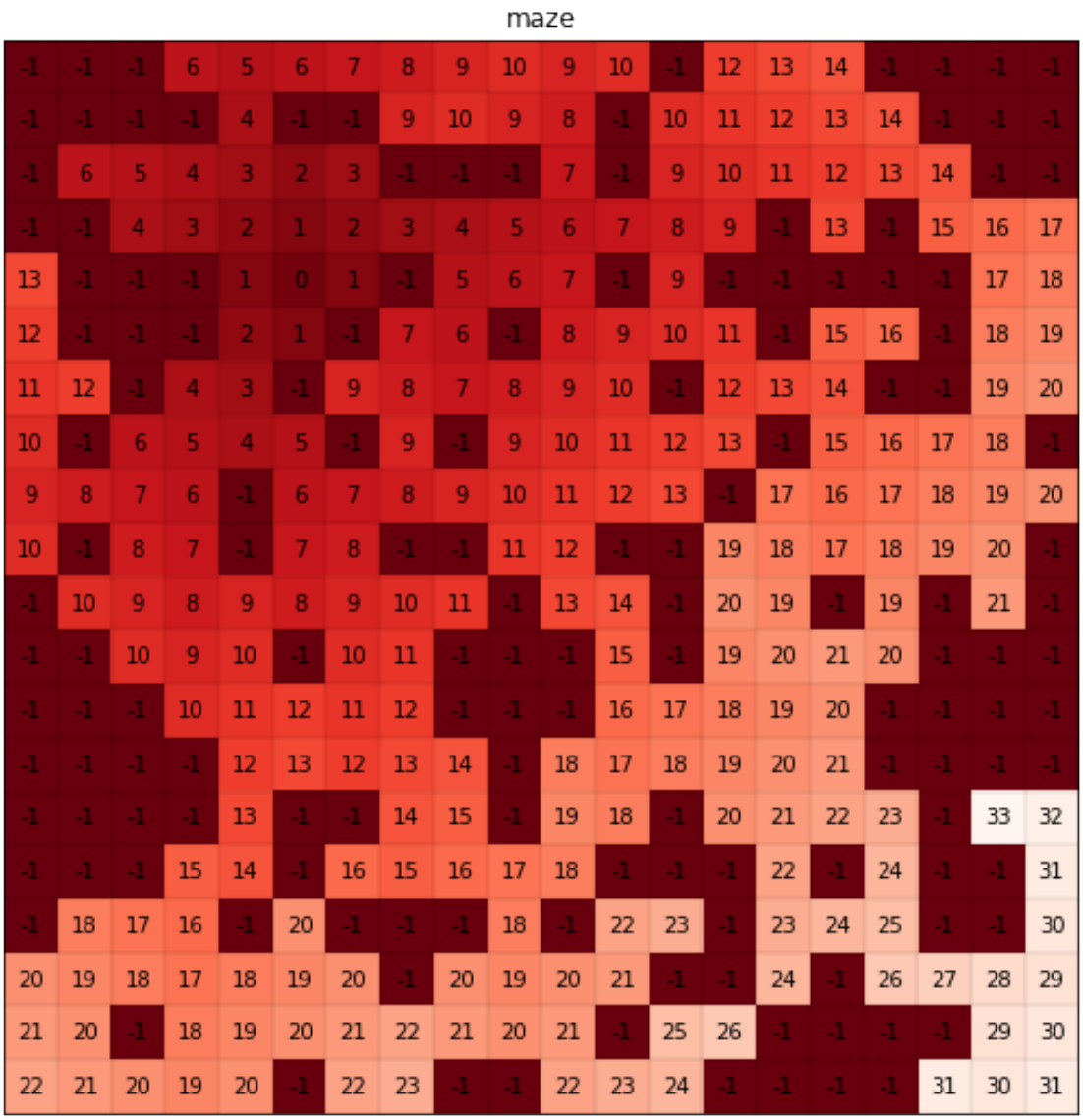
The first step in this process is to create a list, mapping a tuple of x,y coordinates to the worst case fire expansion at a given cell. What we mean by this is that if the fire starts at some arbitrary cell `C`, we can assume the worst conditions for the fire (`q = 1.0`), and calculate for each cell in the grid how many steps it would take for it to reach it. To implement this, we will define a function `generate_fire_step_maze`, which will generate and return this list for a given maze.

We are also going to want a way to render this fire maze for visualization purposes, which is defined by the function `render_fire_maze`. The darker the cell is, the earlier it is on fire (in the worst case of `q = 1.0`) and the lighter the cell is, the later it is on fire.

We can take a look at a visualization of what we are talking about here, by rendering a view of a generated maze and its corresponding `fire_step_map`.

[ ]:
```
maze, fire = gen_fireMaze(20, 0.3)
fire_step_map, fire_step_maze = generate_fire_step_maze(maze)
render_maze(maze,goal=(19,19), fire=True)
render_fire_maze(fire_step_maze, showValues=True)
```


maze




```
<Figure size 600x400 with 0 Axes>
```

At this point we want to modify our A* in order to account for the `fire_step_map`, i.e the future states of the maze. In addition, we will use the manhattan distance to weigh the nodes, this will allow us to explore optimal paths further away from the diagonal. We do all this in the functions `manHat` and `AStar_mod`.

Now we can implement our Strategy 3 in the function `strat_3`.

With strategy 3 now well defined, let us create a method `test_strat3` to test it easily for a given dimension and fire spreading rate. For each test, we will generate the fire start position randomly 10 times and return the number of successes encountered. With our testing method `test_strat3` defined, we can average out probability of success over a defined interval.

2.5 Problem 6

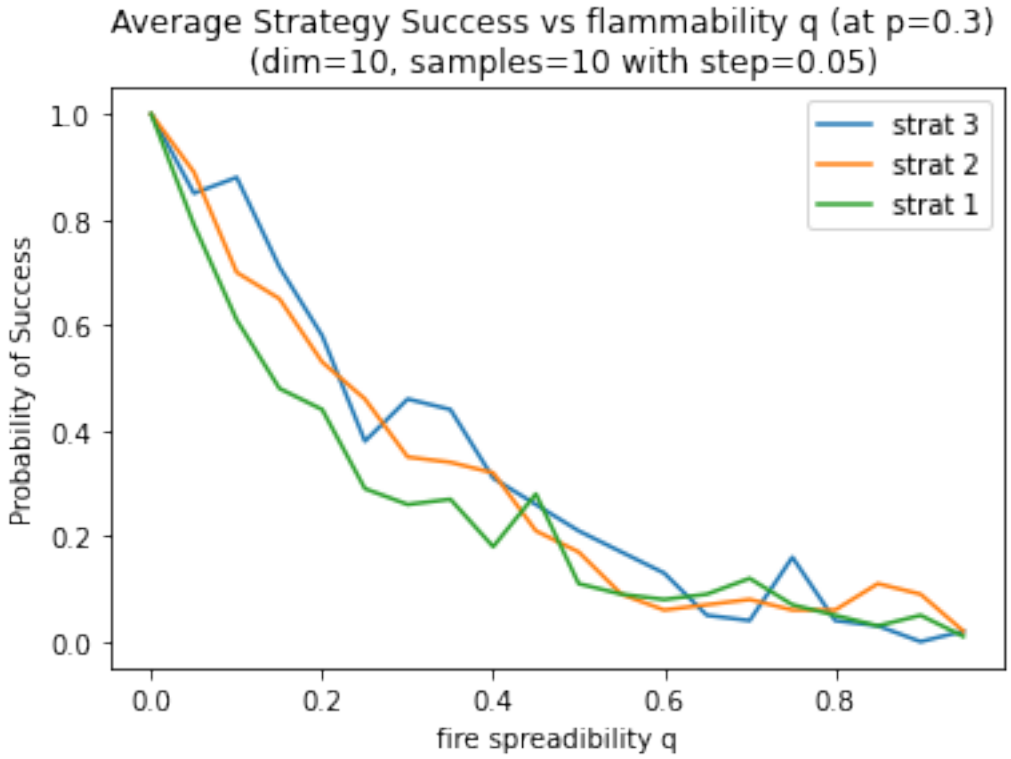
2.5.1 Plot, for Strategy 1, 2, and 3, a graph of 'average strategy success rate' vs 'flammability q' at p = 0.3. Where do the different strategies perform the same? Where do they perform differently? Why?

We can compare each strategy against each other by plotting them on the same graph.

[ ]:
```
# Settings
SAMPLE_COUNT = 20
STEP = 0.05
DIMENSION = 40

# Code to generate graph
densities = np.arange(0.0, 1, STEP).tolist()
successes1 = dict()
successes2 = dict()
successes3 = dict()
density_count = len(densities)
with tqdm(total=density_count * SAMPLE_COUNT) as pbar:
    for i in range(len(densities)):
        q = densities[i]
        if q not in successes2:
            successes1[q] = 0
            successes2[q] = 0
            successes3[q] = 0
        for j in range(SAMPLE_COUNT):
            successes1[q]+= test_strat1(dim=DIMENSION, q=q)
            successes2[q] += test_strat2(dim=DIMENSION, q=q)
            successes3[q] += test_strat3(dim=DIMENSION, q=q)
            pbar.update(1)

x_axis = densities
y_axis = [successes3[x] / (SAMPLE_COUNT*10) for x in densities]
y_2 = [successes2[x]/(SAMPLE_COUNT*10) for x in densities]
y_1 = [successes1[x]/(SAMPLE_COUNT*10) for x in densities]

plt.xlabel("fire spreadibility q")
plt.ylabel("Probability of Success")
plt.title("Average Strategy Success vs flammability q (at p=0.3) \n (dim=" +␣
 ↪str(DIMENSION) + ", samples=" + str(SAMPLE_COUNT) + " with step=" + str(STEP)␣
 ↪+ ")")
plt.plot(x_axis, y_axis, label="strat 3")
plt.plot(x_axis, y_2, label="strat 2")
plt.plot(x_axis, y_1, label = "strat 1")
plt.legend()
```

```
HBox(children=(FloatProgress(value=0.0, max=200.0), HTML(value='')))
```

[ ]: <matplotlib.legend.Legend at 0x7fee2e1a9630>



The three strategies perform around the same at `q >= 0.7`. This is because our Strategy 3 uses the fire map (the worst case future of the fire using `q = 1.0`). At high `q` values such as `0.7` and higher, the worst case fire map that we use in our Strategy 3 A* heuristic is not much worse than the actual fire growth because the actual `q` is very close to `1.0`. Because of this, we are unable to find a viable path to the goal without visiting a node that may be on fire. As a result, our Strategy 3 does about the same as the other two strategies at these high `q` values. However, our Strategy 3 performs better than or equal to the other two strategies at `0.0 <= q <= 0.7`. This is because we assume the worst case fire spread. This means we are assuming a much worse fire spread than the actual fire spread and as a result, the agent chooses a much safer route than Strategies 1 and 2 at these `q` values.

2.6 Problem 7

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

Due to having limited computational resources, our current Strategy 3 only looks at the most optimal paths to the goal cell. We then pass those optimal paths to our risk assessment algorithm and pick the best path. However, if we had unlimited computation resources, our Strategy 3 would instead look at all possible paths to the goal cell. We could then pass every single path to

our risk assessment algorithm which would greatly increase the chance that a viable path to the goal cell is found without catching on fire. For example, if a fire is near a spot that an optimal path may go through, our current Strategy 3 may not be able to escape the fire because it leverages optimal paths. However, with unlimited resources, we could look at all possible paths, therefore avoiding the fire by discovering a less risky but non-optimal path.

2.7 Problem 8

2.7.1 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.

Currently, every time we take a step, we recalculate all the optimal paths using A* and recalculate the worst case fire map using the current fire. This takes a lot time and may take more than 10 seconds between steps. However, if we could only take ten seconds between moves rather than doing as much computation as we like, we would just calculate the next 5 steps and use that to decide which path to take, instead of calculating the entire optimal path. Due to the fact that calculating the entire fire map is extremely quick, we would not need to cut down on how often we recalculate the fire map because we would still be able to do this in less than 10 seconds between moves. As a result, our Strategy 4 would calculate just the next 5 steps using A* rather than calculating the entire optimal path to the goal in between steps. This would greatly cut down how much time we take between moves.