Exploring Search CS 520

Sri Kaavya Toodi, Aravinda Reddy Dandu, Rithvik Reddy Ananth

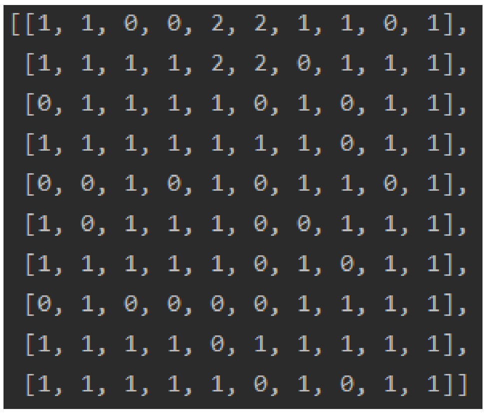
Computer Science Department, Rutgers University-New Brunswick, NJ, US

***Abstract*—Exploring two types of mazes. 1- Using traditional strategies to find a path in a maze with fire propagating from random points. 2- Using A* algorithm to find path in a maze with blocks using different types of heuristics**

I. This *Maze* is on Fire

A. Introduction

The maze will be a square grid of cells. This is represented using a two-dimensional array with integers in Python. Cells can be empty or occupied. An agent starts from the source which is (0,0) and tries to reach the goal (dimension-1, dimension-1). Agent can only move in open cells and in four directions namely up/down and right/left only. But, the maze is on fire and burning down for every step the agent makes. Fire starts at any random cell and propagates with every move of agent. Here, we use 0 to indicate blocked cell, 1 to indicate fire cell, 2 to indicate fire cell. A typical maze in its intermediate state looks like below.
Number of free cells is determined by the probability 'p' which is 0.7 usually.
Rate at which fire spreads is given by q. Probability of a fire spreading to a cell is given by $1-(1-q)^k$ where k is the number of neighbours on fire



```
[[1, 1, 0, 0, 2, 2, 1, 1, 0, 1],
 [1, 1, 1, 1, 2, 2, 0, 1, 1, 1],
 [0, 1, 1, 1, 1, 0, 1, 0, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 0, 1, 1],
 [0, 0, 1, 0, 1, 0, 1, 1, 0, 1],
 [1, 0, 1, 1, 1, 0, 0, 1, 1, 1],
 [1, 1, 1, 1, 1, 0, 1, 0, 1, 1],
 [0, 1, 0, 0, 0, 0, 1, 1, 1, 1],
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 0, 1, 0, 1, 1]]
```

Fig. 1. Typical maze data-structure with fire spreading

Our goal is to find a path for the agent to reach the destination without catching fire. Avoiding obstacles and fire, three strategies are proposed and the efficiency of each strategy is analyzed.

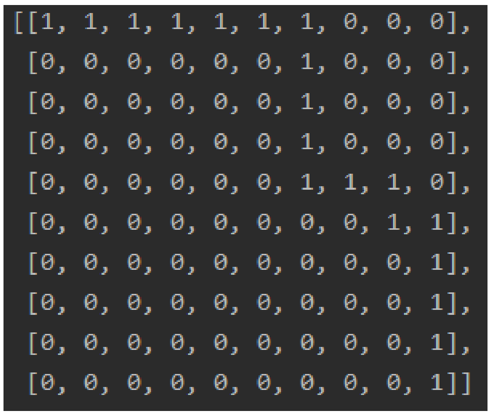
1) Conditions: There are few conditions for a maze to be legal. Only after these conditions are met, the maze will be considered for evaluating a particular strategy.

- For a particular randomly generated maze, there has to be a path from start to end
- Fire can only be put in a location where there is a path from source location to the initial fire location

B. Strategy 1 - Naive

1) Idea: At the start of the maze, wherever the fire is, solve for the shortest path from upper left to lower right, and follow it until the agent exits the maze or burns. This strategy does not modify its initial path as the fire changes.

2) Algorithm implementation: Breadth-first search is used to find a solution from the top left corner to the bottom right corner. In this, path can be constructed by adding a prev attribute to every point traversed. Tracing back from destination gives the path. This functionality is written in *getSolution* method.A sample solution will look like below



```
[[1, 1, 1, 1, 1, 1, 1, 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, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
 [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, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
```

Fig. 2. Solution found by BFS method

After finding a solution, the agent will just traverse this path without considering the movement of fire at all. This naive approach has a minimal rate of success and its pseudocode is below.

Pseudo Code:

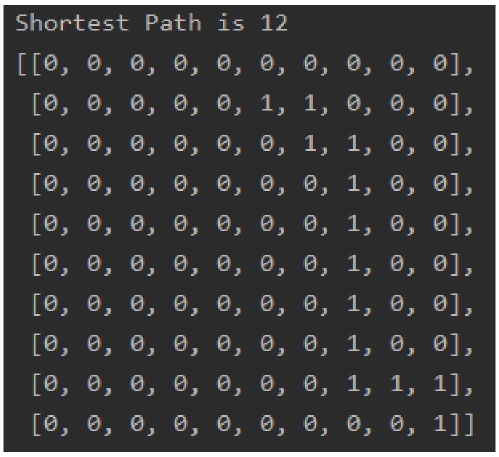
```
 // generate grid with dimensions
maze = generateGrid(dim, fillProb)
solution = initialsolution for grid
firecell = generate a new random point
for point in solution
   maze = spreadFire // Spread fire
   if point catches fire:
        return dead
return alive
```

3) Conclusion: This method takes very little time to execute as a solution is only found once and everything else is just traversal. But chances of survival are low as fire isn't considered. Out of **28589 fair trails**, **5996** times agent has reached the destination. This accounts for **20.7%** survival rate for this strategy

C. Strategy 2 - Rebuild

1) Idea: Strategy 2 is to rebuild the path at every step instead of blindly following a single path to destination. This way *fire* is taken into consideration.

2) Algorithm implementation: The same BFS is used to find a solution but it's found at every single step the agent makes. As fire is considered as a block in the path, the current state of fire is taken into consideration and the agent will try to prevent going in that direction.



```
Shortest Path is 12
[[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, 1, 1, 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, 0, 0, 1, 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, 0, 0, 1, 1, 1],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
```

Fig. 3. Strategy 2 at work

Psuedo Code:

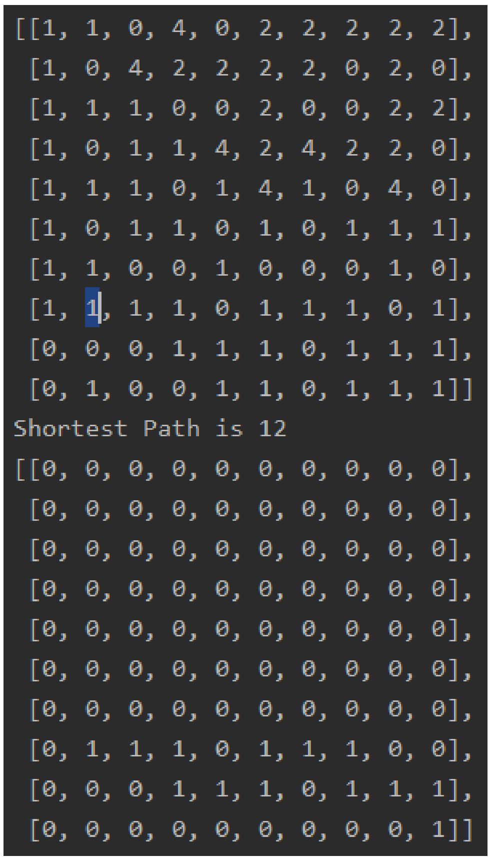
```
// generate grid with dimensions
maze = generateGrid(dim, fillProb)
solution = initialsolution of grid
firecell = generate a new random point
while not in fire
    // Spread fire normally
    maze = spreadFire(normal)
    solution = solution from current point
    if no solution:
      return dead
    point = solution[1]
    if point is destination:
      return alive
```

3) Conclusion: As we try to avoid fire by changing path every step, winning probability is increased substantially. Out of **28470 fair trails**, **8331** times agent has reached the destination. This accounts for **29.26%** of survival

D. Strategy 3 - ***Fake it till you make it***

1) Idea: In the second strategy, we are taking the current state of fire into consideration, but not the future state of it. To consider the future state, we've to estimate where the fire can be after a few iterations and must try to avoid those cells.

2) Algorithm implementation: To find what the future state of fire will be, we introduced a concept to fake the fire for a number of iterations. A new variable is introduced to indicate fake fire and this is used to find a path avoiding the actual cells on fire and also cells that can potentially catch fire.
Fake fire is always spread with probability 1. Heat is the new variable introduced to store the number of times fake fire is to be spread. Until a solution is found, heat is gradually reduced. If heat becomes zero, to have some foresight, it's again made 1. Fire is put off every time after trying to find a solution. If a solution is not found with fake fire being spread, one more try with no fake fire is made. Heat is fixed at the dimension of the maze initially. But while solving, it's found that dimension/2 is enough to give the same result.
Below, an intermediate state with fake fire spread is shown along with a solution from the current path to destination. Here *4* represents fake fire which will be eventually be put off. We can see that all neighboring cells to 2 are put on fake fire with heat=1. We can also see that path taken is avoiding the top right corner.



```
[[1, 1, 0, 4, 0, 2, 2, 2, 2, 2],
 [1, 0, 4, 2, 2, 2, 2, 0, 2, 0],
 [1, 1, 1, 0, 0, 2, 0, 0, 2, 2],
 [1, 0, 1, 1, 4, 2, 4, 2, 2, 0],
 [1, 1, 1, 0, 1, 4, 1, 0, 4, 0],
 [1, 0, 1, 1, 0, 1, 0, 1, 1, 1],
 [1, 1, 0, 0, 1, 0, 0, 0, 1, 0],
 [1, 1, 1, 1, 0, 1, 1, 1, 0, 1],
 [0, 0, 0, 1, 1, 1, 0, 1, 1, 1],
 [0, 1, 0, 0, 1, 1, 0, 1, 1, 1]]
Shortest Path is 12
[[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],
 [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, 1, 1, 0, 1, 1, 1, 0, 0],
 [0, 0, 0, 1, 1, 1, 0, 1, 1, 1],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
```

Fig. 4. Fake fire at work

Psuedo Code:

```
// generate grid with dimensions
maze = generateGrid(dim, fillProb)
solution = initialsolution for grid
firecell = generate a new random point
point = First point in solution
heat = dimension of maze
while maze[point.x][point.y] not in fire
        // Spread fire normally
    maze = spreadFire(normal)
    while solution == 0 and heat > 0:
        spreadFakeFire(maze, heat, 1)
        solution = Sol from current point
        heat = heat - 1
    maze = put off fake fire
    if solution == 0:
        solution = Get solution
        if no solution
         return dead
         break
    point = First point in solution
    if point == dest:
        return alive
return dead
```

3) Conclusion: This method proves to be better than strategy 2. As we're taking potential future state of fire into consideration, chances of survival increase. Out of **41370 fair trails**, **13247** times agent has reached the destination. This means **32.07%** chance of survival which is 3% better than strategy 2.

E. Output Graphs

For each randomly generated maze, 10 simulations are done with 10 different fire locations and 500 such mazes are solved to get better accuracy of average survival chances. For strategy 3, to get even better accuracy, 1000 mazes are solved. These iterations are repeated once for each flammability value q. More the flammability, the lesser the chances of survival. The graph is plotted with flammability against, chances of survival(winning probability). We can see below, strategy 3 has an edge over strategy 2 for almost all the values of flammability. For strategy 1, as it is faster, the maze size can be until 50. For

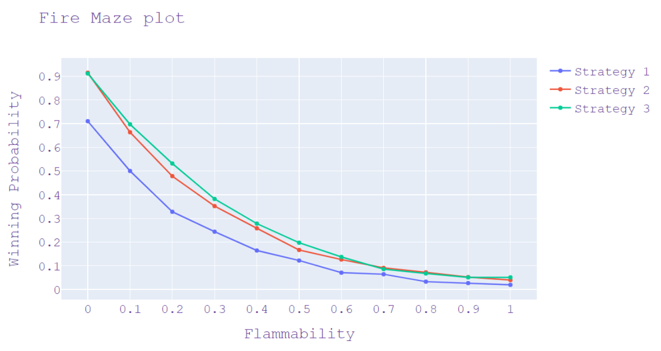


Fig. 5. Graph plotting Winning probability against Flammability

II. Maze Thinning

A. Introduction

In any maze solving problem, a heuristic approach would reduce to no. of nodes visited before we reach the goal node. We have tried implementing geometric and relaxation-based heuristics in an attempt to understand them better. As a classic example of geometric heuristic, we have used Manhattan distance as a heuristic; and for relaxation strategy-based heuristic, we attempted compared obstacle removal relaxation and path direction relaxation.

B. Manhattan distance

1) Idea: Manhattan distance gives us the least path length from a node to the goal node, making it a lower bound of the actual path. So, we can pass this as a heuristic to the A* algorithm.

2) Algorithm implementation:: We have taken the classic Manhattan distance from each node to the goal node and passed it as a heuristic to A* algorithm.

Psuedo Code:

```
# Initialize both open and closed list
    open_list = []
    closed_list = []
# Add the start node to open list
Add Start_Node in the open_list
#while open list is not empty find the path
while the open_list is not empty
    # Get current node
    let the CurrNode equal to first node
    in open_list
    remove the CurrNode from the open_list
    add the CurrNode to the closed_list
    # when you find the goal
    if CurrNode is the goal
        Trace back the path
    # Generate children
    let the children of the CurrNode equal
    the adjacent nodes

    for each child in the children
```

```
# Child is on the closed_list
if child is in the closed_list
    continue to beginning
    of for loop
# Create the t(total path length),
c(source to current node length),
and h (estimated value to goal
from current node) values
child.c = CurrNode.c + distance
between child and current
child.h = distance from child to enc
child.t = child.c + child.h
# if Child is already in open_list
if child.position is in the open list
if child == open_node and
child.c > open_node.c:
    continue
open_list.append(child)
```

```
[[0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 0, 0, 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, 0, 0, 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, 0, 0, 0, 0, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 0, 0, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 0, 0]]
```

Fig. 6. Maze as a 2*2 matrix with path traced along 0's

3) Conclusion: We can repeatedly solve a 1500*1500 matrix under 10 seconds. Now we apply the A* thinning algorithm to solve the same matrix.

C. Obstacle Removal Relaxation

1) Idea: Maze thinning/Obstacle Removal is a relaxation strategy for the maze problem, where we remove a portion of the obstacles in the maze and find the optimal path from the source node using the thinned maze as heuristic. Here we generate a maze the same way as above but we take 0 as free cells and 1 as an obstacle.

2) Algorithm implementation: We remove a fraction (rho) of obstacles from the original maze and apply the A star algorithm at each step to find the distance from the current node to the goal and use this distance as a heuristic for the A*-thinning algorithm. The distance from any given point in the maze to the goal is going to be a lower bound of the distance from source to goal in the original maze as a thinned maze contains more feasible paths. If we save the heuristic values calculated then, we won't have to recompute them in each iteration. **This has reduced the time cost by 10 folds**

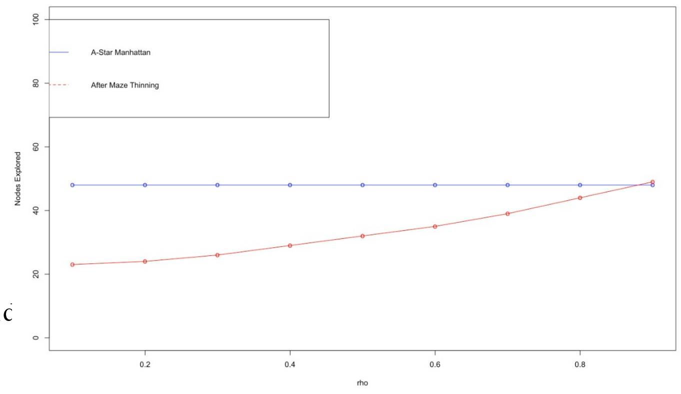


Fig. 7. Graph plotting Nodes explored for Rho values

3) Conclusion: On an average 48 nodes were explored for a 10*10 maze with 30 percent obstacle density. As fraction(rho) of the obstacles are removed, we find that the number of nodes explored are comparatively less. However, as the rho value increases the number of nodes explored also increases. There was an additional 10ms increase in computational time for solving a 10*10 matrix using the A* thinning method. Heuristics can be easily computed with thinner mazes but the number of nodes explored also increases, so it is more viable for lower rho values but does not suit for higher values of rho.

D. Path direction Relaxation

1) Idea: The path relaxation strategy in the maze problem is to relax the restriction of movement of the player to just up, down, left, and right but also allow him to traverse diagonally in the maze.

2) Algorithm implementation: By relaxing the path movement of the player, we now find the distance from the current node to the goal node at every step and use this distance as a heuristic in the A* algorithm to find the path. As the distance traveled diagonally is always lesser, it is going to be a lower bound of the distance from source to goal in the original maze.

3) Conclusion: This is not a viable approach as the number of nodes explored is greater and the average time to solve each maze is also greater (by 100ms than A* Manhattan). For a 10*10 matrix of 1000 mazes we get: Average Nodes explored after path relaxation: 60, Average Time: 0.02003. Irrespective of the representation of the maze factor it is not a viable approach. Manhattan distance provides a meaningful heuristics of a maze and it is a potential solution to approximate a maze.

III. Source code

Source code can be accessed in GitHub

Contributions

Aravinda: Maze on fire all 3 strategies, give an idea on how to approach on maze thinning, and documentation of the maze on fire part.

Rithvik: Implementation of maze thinning, heuristics calculation, plotting graphs, and documentation.
Kaavya: Reduce the time cost of A* thinning, Implementation of maze thinning, give an idea on strategy 3 of the maze on fire, and documentation.

Acknowledgment

We would like to thank Professor Cowan W. Charles and fellow students who have clarified doubts on the discussion board and helped us understand the problem statement better.

References

[1] Class notes
[2] Discussion board on Canvas.
[3] https://stackoverflow.com/questions