REPORT
on

GHOSTS IN THE MAZE

By

ZEESHAN AHSAN
Rutgers University
MS CS
Net ID – za224

ENVIRONMENT DESIGN

I have created a 51 x 51 grid using numpy in python. The maze is made up of '0' s and '1' s. Here, the '0' s are used as walls and '1's are used as free path in the maze/grid.

Initially the maze/grid is just made up of '1's and I create a 51x51 numpy-2D array of all '1's. This is done in the ***getMaze()*** function in the code. After that I iterate over each cell in the 2D array and call the ***getWalls()*** function. The ***getWalls()*** function is used to assign each cell as a wall or as free path in the maze.

In the ***getWalls()*** function:

1. I make a list of 28 '0's and 72 '1's.
2. I choose a random index from 0 to length of my list.
3. I return the value at the above index from this list.

Now, as I iterate through all the cells in my 2D array, I call the ***getWalls()*** function to assign it either as a wall (achieved when ***getWalls()*** return '0') or as a free path(achieved when ***getWalls()*** return '1').

Checking the Validity of the maze:

The validity of the above generated mazes is checked in the ***checkvalid(maze)*** function. Here, I pass the generated maze as the argument of the function. As the first part of the check, I check if the (0,0) position and the (50,50) position are '1's in the maze or not. If they are both '1's, then the Start and the End positions of the maze are unblocked and I go ahead and check if there is a path from start(0,0) to end(50,50) or not.

Grey Question1 Answer: Here, I have gone with the ***Depth First Search (DFS)*** approach to check if an unblocked path from start (0,0) to end (0,0) exists or not. I chose DFS over BFS here as I just had to find any path from start to end and not the shortest path (BFS Path). Another consideration I took was that the compute cost in DFS is low than BFS. So, I went with DFS in this instance. For this, I call the ***dfsSearch(maze, start)*** function. Here, the arguments are the maze and the start cell of the maze. The ***dfsSearch(maze,start)*** is implemented as follows :

1. Create a ***stack***.

2. Create a ***visited list***.

3. Take the first cell (start) and append/push it in the stack.

4. We run a ***while loo*p** till the stack is not empty:
   a. We take the ***current Cell(x,y)*** from the stack and check if it is – (50,50) or not.

   b. If it is (50,50) we return ***'True'*** to the caller which signifies that we have reached to the end cell (50,50) of the maze and hence there is a path from start cell to end cell of the maze and the maze is valid.

   c. Otherwise, we check the right of the current cell (given that there is a cell on the right – this is checked by: if(y+1) <= 0) and if the right cell is not a wall. (this is checked by maze[x][y+1] != 0)

   d. We check if this cell is in the visited list or not.
      a. If it is in the visited list, we don't go to that cell again.
      b. Otherwise, we move there and append it to the stack.

   e. We do this for all valid neighboring cells to the current cell. (i.e. – cell to the right of the current cell, cell to the left of the current cell, cell down of the current cell and cell up to the current cell).
   (*I am not mentioning the logic for each check as they are like the above point, and it will be repetitive, and the report has to be concise.*)

   f. Then at the end of the loop body we ***pop()*** the last appended cell to the stack and continue with our search from there.

   g. If we don't reach (50,50) till all the elements from the stack are exhausted, we don't return ***True*** and this signifies that there is no valid/unblocked path between start and finish.

Automating the creation of mazes:

For this, I have created a helper function ***getValidMaze().***

In the ***getValidMaze()*** function, I run a ***while(True)*** loop and generate a new maze, then I call the ***isValidMaze(maze)*** with this maze as argument. If the ***isValidMaze(maze)*** function returns true, then I return the valid ***maze*** . Otherwise, I create another maze and check for its validity. I do this till I get a valid maze and return it from the ***getValidMaze()*** function.

GHOST IMPLEMENTATION

For the implementation of ghosts in the project, I have used a dictionary in python. For this, I have created a function ***getGhosts(n)*** which takes the number of ghosts to be generated in the maze as input. Inside this function, I create a dictionary named ***ghosts.*** Then,

1. I run a for loop till ***n (number of ghosts)*.**

2. Inside the for loop, I generate 2 random number between 0 and 50.

3. If these are (0,0) or (50,50) – designating the start cell and the end cell, I generate another set of random numbers between 0 and 50.

4. Then I assign it to the dictionary ***ghosts*** – ***with key as the for loop variable***, and the ***value as the tuple of the random numbers*** generated in the previous step. The sample dictionary looks like:
   Ghosts: {1: (47, 2), 2: (43, 29), 3: (23, 45), 4: (49, 16), 5: (24, 27), 6: (8, 1), 7: (20, 17), 8: (20, 30), 9: (5, 48), 10: (14, 2)}.

5. I return this dictionary from the function to get a new set of ghosts every time.

Here I observed that since ghosts can move through walls. They will never be walled off. So, I can spawn the ghosts at random cells, even walls.

GHOST MOVEMENT

The ghost movement is achieved by the ghost movement function. This function ***ghostMovement (ghostDict, maze)*** takes 2 input arguments – (a) The ghost dictionary , (b) The maze. This function returns a new ghost dictionary with the modified cell positions for each key.

For the movement of each ghost, I iterate over the ghost dictionary and access the cell values (the values in the dictionary).

Here I have used 4 values, generated randomly, namely – 1,2,3,4.
These values designate where the ghost moves.

- If the value is 1, then ghost moves right.
- If the value is 2, then ghost moves left.
- If the value is 3, then ghost moves up.
- If the value is 4, then ghost moves down.

For each value in the ghost dictionary, I do the following:

1. If the value is (50,50) – End Cell. Then only 2 moves can be made by the ghost. It can go up or left. So, I choose a random number from the list of [2,3].

2. If the value is (0,0) – Start Cell. Then only 2 moves can be made by the ghost. It can go down or right. So, I choose a random number from the list [1,4]

3. If the value is (50,0) – Last row, 1st column. Then again 2 moves are possible. Either up or right. So, I choose a random number from the list [1,3].

4. If the value is (0,50) – First row, 50th column. Again 2 moves are possible. Either left or down. So, I choose a random number from list [2,4].

5. If the value is (**not** 0,50) – Ghost in the last column and not first row. Then 3 moves are possible – Left, up or down. So, I choose a random number from the list [2,3,4].

6. If the value is (**not** 0,0) – Ghost in 1st Column and not first row. Then 3 moves are possible – Right, up and down. So, I choose a random number from the list [1,3,4].

7. If the value is (0, **not** 0) – Ghost not in 1st column and ghost in 1st row. Then 3 moves are – Right, Left and down. So, I choose a random number from the list [1,2,4].

8. If the value is (50, **not** 0) – Ghost in last row but not 1st Column. Then 3 moves are – Right, Left, up. So, I choose a random number from the list [1,2,3].

9. If it is not these cases, then the ghost is free to move in any direction. So, I choose a random number from [1,2,3,4].

10. Now I must check for walls in each move and assign (0.5 probability) if it is a wall. I do this by:

    a. If the move cell is wall, then choose randomly between 2 numbers from a list [1,2], if I get the number '1' , -> I move to the wall else I stay in place.

After doing the above steps, I return the ghost Dictionary with the new cells for each ghost.

AGENT 1

For Agent 1, I plan the shortest path using the Djikstra Algorithm to find the shortest path. I have gone with Djikstra Algorithm because I wanted a more efficient search technique than BFS for shortest path as I have to find shortest path in many places for future agents.

A brief on how I have implemented Djikstra Algorithm:

1. I make a dictionary of unvisited cells and instantiate the cost of each to infinity, I also make an empty dictionary of visited cells.
2. Then I update the cost for the start cell as 0.
3. I make the ghost set from the dictionary of ghosts to check if the current cell is in the ghost Set.
4. Then I go through the cells while the unvisited dictionary is not empty.
    a. I always update the current cell as the minimum of the unvisited cells based on their cost.
    b. I add the value of unvisited [current cell] to the key visited[current cell] in the visited dictionary.
    c. I check, if the current cell is (50,50) and if (50,50) is in the reverse path keys, I build a reverse path and return it in a list to the caller of the function. If (50,50) is not in the keys, that means no reverse path is possible from goal to start and in this case I return (-2) to the caller.
    d. Then I build a valid neighbors list of the current cell. Then for each valid neighbor in this list:
        i. If neighbor is in visited dictionary, I continue to the next iteration as we don't need to do anything here.
        ii. Otherwise, I calculate the distance to travel here.
        iii. If this distance is less than previous value in the unvisited dictionary for this key (neighbor key), I update this value for that key in the dictionary of unvisited.
    e. I pop / remove the current cell from the unvisited dictionary.

For Agent1, I call the function ***djikstra_ag1(maze,ghostDict).*** In this method I pass the maze and ghost dictionary as arguments.

I call the djikstra path calculating function with start as cell(0,0) and an empty dictionary of ghosts which represents the path with no ghosts which is the requirement for agent 1.

I store this path in the path variable which is a list.

Now I run a while loop till I see the cell(50,50).
Inside the while loop:

1. I assign the current cell as the last element of the path. This is the first cell for forward movement, as I always get a reverse path from djikstra function.

2. Now, I make a new ghost Set to check for the ghosts and current cell.
    a. For this, I iterate over every dictionary item using and make a set of ghost cells.

3. I check if current cell is in the ghost set -> Then, return (-2) which denotes that the agent has died.

4. Otherwise, I check if the cell is (50,50) (end cell). If it is the end cell -> I return (100), denoting that the agent has reached its goal cell.

AGENT 2

I have implemented agents 2 in the function ***djikstra_ag2(maze, start,ghostDict).*** Here the arguments are maze – The maze object, start – The start cell from where path to goal must be calculated, ghostDict – Dictionary object for ghost cells.

I implement agent 2 as follows:

1. Build a ghost Set and check if the start cell is in the ghost Set or not ?
    a. If the start cell is in ghost set -> return (-2), indicating the agent has died.
    b. If not go to step 2.

2. While current cell is not (50,50). I do the following:
    a. I calculate a path to the goal cell from current cell with the current configuration of ghosts in the maze.
    b. ***While*** there is no path to the goal cell, I do the following:
        i. *I calculate the nearest ghost using the function* ***nearestGhost(ghostSet, curCell).***
        ii. *I check each Valid possible neighbor of current cell (i.e.,* ***the neighbor cell is within the maze and the neighbor cell is not a wall and the neighbor cell is not a ghost****) and append it to the neighborList. For every cell, this can be of max Length = 4.*
        iii. *If there are no valid possible neighbor cells(i.e neighborList is Empty) then, I move the ghosts and hope for the best result.(i.e some neighbor cell to free up so that the agent can move there).*
        iv. *If there are valid neighbors present (i.e., length of neighborList is not zero), then for each neighbor in the list I find the distance from the nearest ghost obtained in step (i) .'*
        v. *I move to the cell farthest from nearest ghost.*
        vi. *I move the ghosts, clear and update my ghostSet and check if the agent has died because of this movement. If it has died -> I return (-2) which denotes agent is dead.*
        vii. *I calculate a new path again from this new cell. If there is a path to the goal from here, I exit the while loop in step 2,*

otherwise I repeat the above steps till I find a valid path to goal cell or my agent 2 dies.

3. I pop the last element from the reverse path and move there.

4. I move my ghosts, clear and update my ghostSet and check if agent has died. If it has died, I return (-2).

5. I check if the current Cell is (50,50). If it is (50,50) I return (100) denoting the agent has reached the goal. I do these till my agent dies or I reach the goal cell.

Grey Question2 Answer: We don't always need to replan. Till we have a valid path from the current cell to the goal cell with the current configuration of ghosts, we can move in that path. When I see that I can't reach the goal from current cell, I must replan a new path and start executing the moves in this path.

AGENT 3

I have implemented agent 3 in the function ***djikstra_ag3(maze, start, ghostDict)*** **–** Here I pass the maze, start cell and the ghost Dictionary as the input arguments.

I do the following steps to attain agent 3 behavior:

1. I build the ghost set.

2. I instantiate counters 5 counters cVal, rVal, lVal, dVal, uVal for survivability in simulations from current cell, valid right neighboring cell from current cell, valid left neighboring cell from current cell, valid down neighboring cell from current cell and valid up neighboring cell from current cell respectively.

3. I do the following till I reach the cell (50,50):
    a. *Run a simulation of agent 2 using the current configuration of ghosts from the current cell. I increment the survivability counter for current cell if the simulation is successful. (i.e., the agent2 reaches (50,50))*
    b. *Run a simulation of agent 2 using the current configuration of ghosts from a valid right neighbor of current cell. I increment the survivability counter for right cell if the simulation is successful. (i.e., the agent2 reaches (50,50))*
    c. *Run a simulation of agent 2 using the current configuration of ghosts from the valid left neighbor of current cell. I increment the survivability counter for left cell if the simulation is successful. (i.e., the agent2 reaches (50,50))*
    d. *Run a simulation of agent 2 using the current configuration of ghosts from a valid down neighbor of the current cell. I increment the survivability counter for down cell if the simulation is successful. (i.e., the agent2 reaches (50,50))*
    e. *Run a simulation of agent 2 using the current configuration of ghosts from a valid up neighbor of current cell. I increment the survivability counter for current cell if the simulation is successful. (i.e., the agent2 reaches (50,50))*
    f. *I find the maximum value from these survivabilities.*

g. *I make the move with the maximum survivability.*
h. *If survivability is zero in all directions -> I still take the move which makes me move towards the goal. (I did this after pondering and experimenting a lot. One strategy was to move away from nearest ghost, but often the agent goes back to his previous position or moves away from the goal. This move increases survivability of the agent, but he never reaches or moves towards the goal.) So, I thought to make him move in the direction of the goal if that move is possible (i.e., down neighbor cell or right neighbor cell is free) as this moves me closer to the goal).*
i. *After making a move, I move the ghosts and check if they killed the agent.*
j. *I do the above steps till I reach the goal cell (50,50) or the agent dies.*

Grey Question3 Answer: With agent 3, when the projected survivability is zero from all neighboring cells and from the current cell, then there are 3 strategies to choose from – (a) Stay in place and hope for the best, (b) Move away from the nearest ghost, if possible (i.e., neighboring cells have a valid movement possible), (c) Move in the down direction or right direction from the current cell as these move me closer to the goal cell. Here, after running simulations and experimenting a lot I chose to go with the 3rd option as I observed that in the 3rd agent, he keeps moving in some local neighboring cells and never reaches the goal cell. He gets similar survivability values from neighboring cells very often and moves very slowly. It was not possible for me to get a result in which it survived and reached the goal ever in my experiments. As a result, I chose a kind of (heuristic) which always takes him towards the goal if possible.

No, theoretically survivability zero from all possible moves does not guarantee that success is impossible as I am running just simulations from the current position and none of those moves are taking place in the actual maze for agent 3. The ghost movement will be different for all simulations as it has a high degree of randomness about it. So, it is not guaranteed that success is impossible in that case in agent 3 as there might be a case in the actual agent 3 movement where ghosts move in such a random manner that agent reaches the goal cell.

AGENT 4

I have implemented agent 4 in ***djikstra_ag4(maze, start, ghostDict)*** function. Here the arguments are maze, the starting position and the ghost dictionary for ghosts.

I had observed that even though theoretically agent 3 is very intelligent(as it always choses the best move possible for it with highest survivability value); I observed that this in fact is not a very good heuristic and leads to very slow movement of the agent and ultimately results in very high survivability but extremely slow movement and getting caught in some sort of radius of neighboring cells or death(as ghosts eventually come to his cell and kill him).

As a result, I felt, the need to introduce some sort of efficiency where the agent would move more quickly towards the goal and whenever he is faced with a problem (i.e., no path to goal -> then he would use his intelligence (simulate futures) to make the best move possible and take the move with highest survivability).

I implement this as follows:

1. I make the ghost set and check if the current cell is in the set. If the agent dies, I return (-2)

2. I iterate and do the following steps till I reach cell (50,50):
    a. *I calculate a path using djikstra algorithm from current cell to goal cell using the current configuration of the ghosts.*
    b. *Then if I don't find a path from the current cell to goal I do the following using till I find a path to goal:*
        a. *I make an empty neighbor list and instantiate counters as zero for simulations from current cell and all valid neighbors of current cell.*
        b. *Then I simulate the future movement using agent 2 and get the results for each possible move.*
        c. *If there are no possible neighboring moves (i.e., neighbor list is empty then I decide to stay in place)*

    d. *If moves are possible but the survivability is zero on all moves in the simulation (max of survivability is zero), then I nearest ghost and make the move which takes me furthest from nearest ghost.*
    e. *Otherwise, I take the move with the highest survivability from this and move there.*
    f. *I move the ghosts and check if agent is dead, if agent is dead, I return (-2)*
    g. *Otherwise, I calculate the path again from this cell. If I find a path, I get out from this loop otherwise I repeat the steps of the loop.*

3. I make the next move from the reverse path by popping the last element from the list of reverse path and setting the current cell to that value.

4. I move the ghosts and check if agent is dead, if agent is dead, I return (-2).

5. I check if agent has reached goal cell and return (100), if agent has reached goal cell.

Agents which can't see the ghosts in walls

AGENT 1:

Agent 1 does not change due the fact that it cannot see the ghosts in walls as it never takes the ghosts into consideration when planning the path in the first instance as well. So, any information about ghosts does not matter to him when planning the path.

AGENT 2, AGENT 4:

I have implemented the agent 2, agent 4 which cannot see the ghosts in the ***less_info_ag2(maze, start, ghostDict), less_info_ag4(maze, start, ghostDict)*** method. It takes in as input – the maze, start cell and the ghost dictionary.

To achieve the functionality of agent not being able to see the ghosts in the walls, I have made a new dictionary of ghosts which I pass when agent tries to plan a path and calls the ***djikstra (maze, curCell, agentGhostDict).***

Here, I make the new dictionary of ghosts in which the ghosts which are in walls are not present. I do this as follows:

1. I create a new ghost set from previous ghost set but check if the ghost is not in walls (i.e., maze[ghost[0]][ghost[1]] !=0), then I add to the new ghost set (agentGhostSet), otherwise I don't add it to the new ghost set.
2. I create a new dictionary of ghosts from this new ghost set and pass it whenever the agent has to plan the path.

The rest of the things are same in agent 2 and agent 4 as stated previously.

AGENT 3:

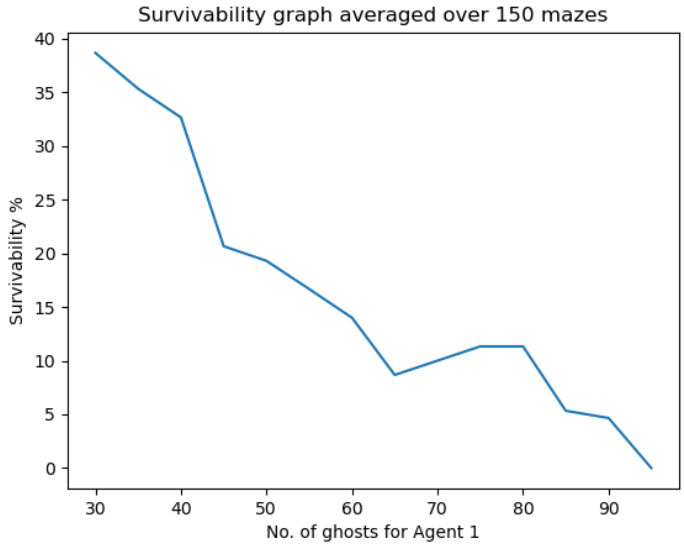
Here, the agent 3 will again fail always as it will face the same problems to a higher degree as it can't see the ghosts in the wall. As a result, it's survivability will

again be zero or it will just always move in a somewhat fixed radius and will never reach the goal cell.

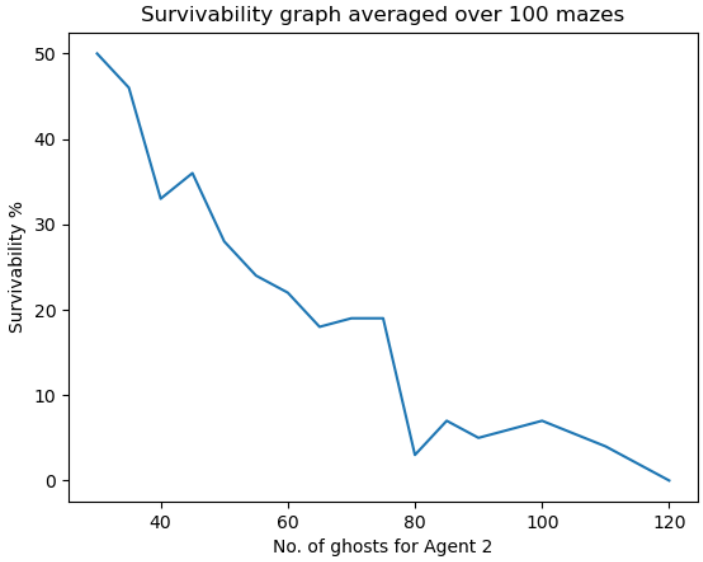
So, I have created the less information agents for only agent 2 and agent 4.

Graphs of Survivability of Agents

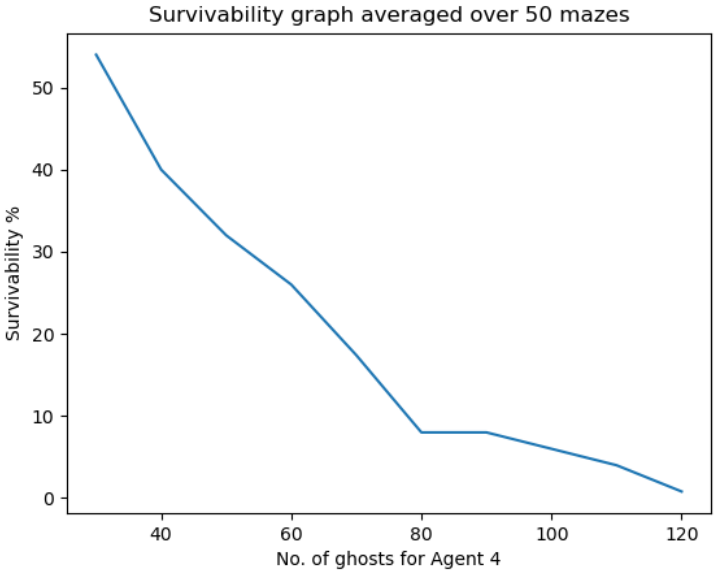
AGENT 1



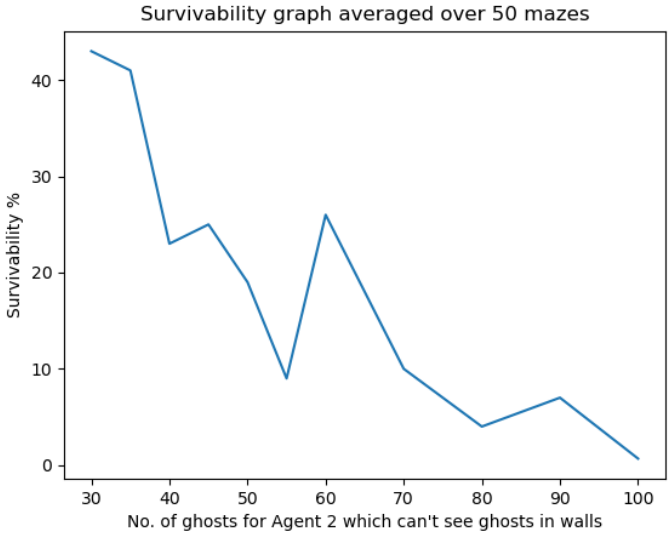
AGENT 2



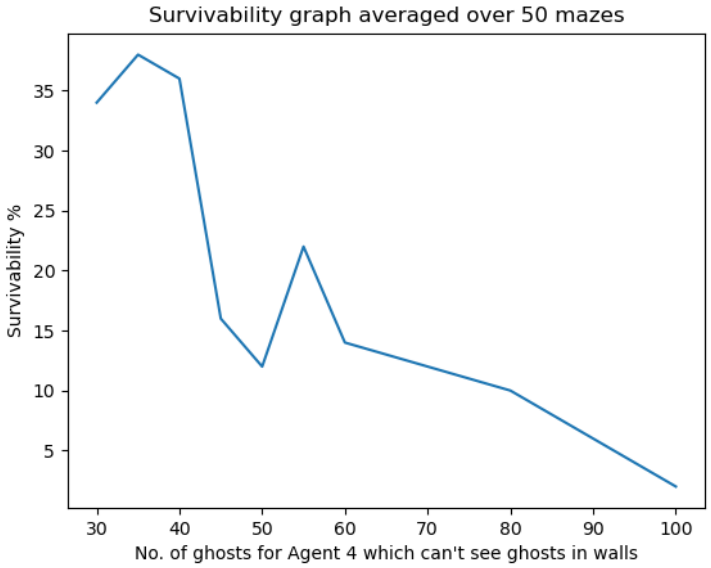
AGENT 4



AGENT 2 Which can’t see ghosts in walls



AGENT 4 Which can’t see ghosts in walls



Computational Bottleneck

In the implementation of the agents, the most difficult was to optimize the agent 3. In that agent, if I run a lot of simulations it does conclude to a better path but lacks highly in efficiency and takes forever to decide for its next step. Also, if it decides, later there is a probability for the agent to go back to the previous cell from where it came and hence face the same issues as before. Eventually, I realized that a tie breaker is required in agent 3 and I always chose the heuristic in which it goes right or down (as these movements take him closer to the goal cell). Still, I was not able to get deterministic survivability values for agent 3.

As for the agent 3 when it can’t see the ghosts in the walls, the situation of its survivability will be worse than before as more ghosts will be probable to kill the agent.

After this, I realized that agent 3 is ‘too smart’ for its own good and never concludes to the final goal.

I kept these things in mind when designing agent 4 and as a result only used simulations ('intelligence') when I had no path to go to the goal cell.

KEY OBSERVATIONS:

1. Agent 2 performs better at almost every instance than agent 1. This is also expected, as agent 1 just plans the path once and starts executing it till it either dies or reaches the goal cell. Initially when the ghosts are less the path tends to be unblocked and unaffected a lot by the ghost movements. So, the survivabilities are somewhat similar in those cases. But, as the ghosts increase, agent 2 outperforms agent 1 as it can make smarter decisions considering the current configuration of the ghosts.
2. Agent 3 performs worse than Agent 2 as it is 'too smart' for its own good. It always runs to computational bottlenecks as it tries to find the best possible move at each step considering only survivability rather than closeness towards the goal cell in each movement. It falls into the trap of going back and forth in some sort of radius and eventually dies by ghost movement.
3. Agent 4 outperforms Agent 1, Agent 2 and Agent 3. This is because it balances efficiency and intelligence optimally. Only when there is no path to the goal cell, it uses its intelligence and simulates the future for possible movements. It chooses the best move from these simulations which will take him to the goal cell. Hence, it's survivability is higher than the preceding agents.
4. When considering agents when they can't see ghosts in walls, Agent 1 does not change in his behavior or survivability as it anyways does not care about ghosts at all when planning a path to goal. So, if there is ghost in the wall or not does not affect him.
5. Agent 2 when he can't se. ghosts in walls, he performs worse than Agent 2. This is also expected as the ghosts it can see and avoid in every movement is reduced. This becomes highly pronounced when

the number of ghosts is high and the probability of them being in walls is higher.

6. Same thing was observed for Agent 4 who can't see the ghosts in the walls. Again, the effects become pronounced when the number of ghosts is high and hence survivability tanks faster towards zero in this case.