Probabilistic Sensing

Advith Chegu (ac1771) & Naveenan Yogeswaran (nry7)

Both group members contributed equally to the code, data collection, and report.

Prior to any interaction with the environment, what is the probability of the target being in a given cell?

Prior to any interaction with the environment, each cell should have equal probability of containing the target. Given the dimension for the grid D, the probability for each cell is $1/(D^2)$.

Let $P_{i,j}(t)$ be the probability that cell (i,j) contains the target, given the observations collected up to time t. At time $t+1$, suppose you learn new information about cell (x,y). Depending on what information you learn, the probability for each cell needs to be updated. What should the new $P_{i,j}(t+1)$ be for each cell (i,j) under the following circumstances:

- At time t + 1 you attempt to enter (x,y) and find it is blocked?

  $P_{i,j}(t+1)$

  $= P(\text{in (i,j) | not (x,y)})$

  $= P(\text{in (i,j) and not (x,y)})/P(\text{not (x,y)})$

  Because the target can only be in one cell, we know $P(in(i,j)andnot(x,y))$ is the same as $P(in(i,j))$.

  $= P(\text{in (i,j)})/P(\text{not (x,y)})$

  $= P_{i,j}(t)/(1-P_{x,y}(t))$

  Answer: For every cell except (x,y), $P_{i,j}(t+1) = P_{i,j}(t)/(1-P_{x,y}(t))$

  For cell (x,y), $P_{x,y}(t+1) = 0$

- At time t + 1 you attempt to enter (x,y), find it unblocked, and also learn its terrain type?

  $P_{i,j}(t+1)$

  $= P_{i,j}(t)$

  Answer: $P_{i,j}(t+1) = P_{i,j}(t)$

- At time t + 1 you examine cell (x,y) of terrain type flat, and fail to find the target?

  $P(\text{failed at (x,y)})$

  $= P(\text{failed at (x,y) and in (x,y)}) + \sum_{i,j} P(\text{failed at (x,y) and in (i,j)})$

$= P(\text{in (x,y)}) * P(\text{failed at (x,y) | in (x,y)}) + \sum_{i,j} P(\text{in (i,j)}) * P(\text{failed at (x,y) | in (i,j)})$

Because the target can only be in one cell, we know $P(failedat(x,y)|in(i,j))$ is 1.

$= P(\text{in (x,y)}) * 0.2 + \sum_{i,j} P(\text{in (i,j)}) * 1$

$= P(\text{in (x,y)}) * 0.2 + (1 - P(\text{in (x,y)}))$

$= P_{x,y}(t) * 0.2 + (1 - P_{x,y}(t))$

New probability for every cell except (x,y).

$P_{i,j}(t+1)$

$= P(\text{in (i,j) | failed at (x,y)})$

$= P(\text{in (i,j) and failed at (x,y)})/P(\text{failed at (x,y)})$

$= (P(\text{in (i,j)}) * P(\text{failed at (x,y) | in (i,j)}))/P(\text{failed at (x,y)})$

Because the target can only be in one cell, we know $P(failedat(x,y)|in(i,j))$ is 1.

$= (P(\text{in (i,j)}) * 1)/P(\text{failed at (x,y)})$

$= P(\text{in (i,j)})/P(\text{failed at (x,y)})$

$= P_{i,j}(t)/(P_{x,y}(t) * 0.2 + (1 - P_{x,y}(t)))$

New probability for cell (x,y)

$P_{x,y}(t+1)$

$= P(\text{in (x,y) | failed at (x,y)})$

$= P(\text{in (x,y) and failed at (x,y)})/P(\text{failed at (x,y)})$

$= (P(\text{in (x,y)}) * P(\text{failed at (x,y) | in (x,y)}))/P(\text{failed at (x,y)})$

$= (P(\text{in (x,y)}) * 0.2)/P(\text{failed at (x,y)})$

$= (P_{x,y}(t) * 0.2)/(P_{x,y}(t) * 0.2 + (1 - P_{x,y}(t)))$

Answer: For every cell except (x,y), $P_{i,j}(t+1) = P_{i,j}(t)/(P_{x,y}(t) * 0.2 + (1 - P_{x,y}(t)))$.

For cell (x,y), $P_{x,y}(t+1) = (P_{x,y}(t) * 0.2)/(P_{x,y}(t) * 0.2 + (1 - P_{x,y}(t)))$.

- At time t + 1 you examine cell (x, y) of terrain type hilly, and fail to find the target?

  Answer: For every cell except (x,y), $P_{i,j}(t+1) = P_{i,j}(t)/(P_{x,y}(t) * 0.5 + (1 - P_{x,y}(t)))$.

  For cell (x,y), $P_{x,y}(t+1) = (P_{x,y}(t) * 0.5)/(P_{x,y}(t) * 0.5 + (1 - P_{x,y}(t)))$.

- At time t + 1 you examine cell (x, y) of terrain type forest, and fail to find the target?

  Answer: For every cell except (x,y), $P_{i,j}(t+1) = P_{i,j}(t)/(P_{x,y}(t) * 0.8 + (1 - P_{x,y}(t)))$.

  For cell (x,y), $P_{x,y}(t+1) = (P_{x,y}(t) * 0.8)/(P_{x,y}(t) * 0.8 + (1 - P_{x,y}(t)))$.

- At time t + 1 you examine cell (x, y) and find the target?

  Answer: For every cell except (x,y), $P_{i,j}(t+1) = 0$.

  For cell (x,y), $P_{x,y}(t+1) = 1$.

At time t, with probability $P_{i,j}(t)$ of cell (i, j) containing the target, what is the probability of finding the target in cell (x, y):

- If (x, y) is hilly?

  P(finding target in (x,y))

  $= P$(in (x,y) and finding in a hilly terrain) $\leftarrow$ These two are independent events

  $= P$(in (x,y)) $* P$(finding in hilly terrain)

  $= P_{x,y}(t) * (1 - 0.5)$

  $= P_{x,y}(t) * 0.5$

  Answer: $P_{x,y}(t) * 0.5$

- If (x, y) is flat?

  P(finding target in (x,y))

  $= P$(in (x,y) and finding in a flat terrain) $\leftarrow$ These two are independent events

  $= P$(in (x,y)) $* P$(finding in flat terrain)

  $= P_{x,y}(t) * (1 - 0.2)$

  $= P_{x,y}(t) * 0.8$

  Answer: $P_{x,y}(t) * 0.8$

- If (x, y) is forest?

  P(finding target in (x,y))

  $= P$(in (x,y) and finding in a forest terrain) $\leftarrow$ These two are independent events

  $= P$(in (x,y)) $* P$(finding in forest terrain)

  $= P_{x,y}(t) * (1 - 0.8)$

  $= P_{x,y}(t) * 0.2$

Answer: $P_{x,y}(t) * 0.2$

- If (x, y) has never been visited?

  P(finding target in (x,y))

  $=$ P(in (x,y) and finding in an unknown terrain) $\leftarrow$ These two are independent events

  $= P$(in (x,y)) $* P$(finding in unknown terrain)

  $= P$(in (x,y))$*$(P(finding in terrain and terrain is blocked)$+P$(finding in terrain and terrain is flat)$+$ P(finding in terrain and terrain is hilly)$+P$(finding in terrain and terrain is forest))

  $= P$(in (x,y))$*$(P(terrain is blocked)$*P$(finding in terrain | terrain is blocked)$+$ P(terrain is flat)$*P$(finding in terrain | terrain is flat)$+P$(terrain is hilly)$*$ P(finding in terrain | terrain is hilly)$+P$(terrain is forest)$*P$(finding in terrain | terrain is forest))

  $= P$(in (x,y)) $* ((0.3) * (1 - 1) + (0.7) * ((1\,/\,3) * (1 - 0.2) + (1\,/\,3) * (1 - 0.5) + (1\,/\,3) * (1 - 0.8)))$

  $= P$(in (x,y)) $* (0.7) * ((1\,/\,3) * 1.5)$

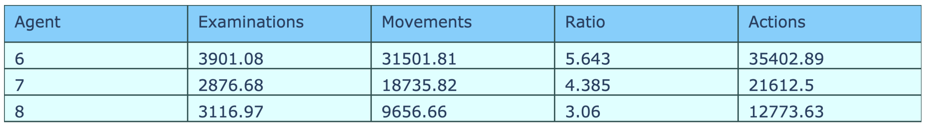
  $= P$(in (x,y)) $* (0.7) * 0.5$

  $= P_{x,y}(t) * 0.35$

  Answer: $P_{x,y}(t) * 0.35$

Implement Agent 6 and 7. For both agents, repeatedly run each agent on a variety of randomly generated boards (at constant dimension) to estimate the number of actions (movement + examinations) each agent needs on average to find the target. You will need to collect enough data to determine which of these agents is superior. Do you notice anything about the movement/examinations distribution for each agent? Note, boards where the target is unreachable from the initial agent position should be discarded.

The following tables and graphs are based off the data collected by running each agent on 100 different 50x50 gridworlds.



Agent	Examinations	Movements	Ratio	Actions
6	3901.08	31501.81	5.643	35402.89
7	2876.68	18735.82	4.385	21612.5
8	3116.97	9656.66	3.06	12773.63

Figure 1: Averages Table

As shown in the table above, we can see that Agent 7 is superior over Agent 6 as Agent 7 performs less actions on average than Agent 6. This is likely because Agent 7 takes the terrain type and their false negative rates into consideration when deciding which cell to examine next. As a result, Agent 7 will prioritize cells where it'll likely find the target over cells that just may contain the target.

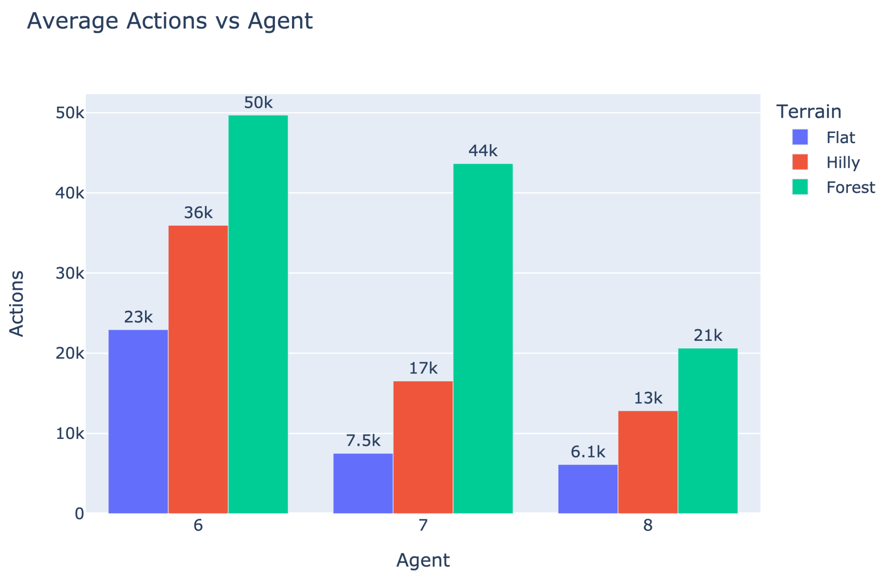


Figure 2: Bar Chart

As shown in the bar graph above, both Agents seem to take more actions as the false negative rate increases which makes sense as higher false negative rates increases the chances of failing an examination which increases the number of actions needed to reexamine the cell later. Agent 7 seems to still consistently beat Agent 6 among the various terrain types as well. Although Agent 7 does struggle more when the target is in a forest, the way the agent prioritizes finding the cell allowed it to still beat Agent 6 on average for every terrain.

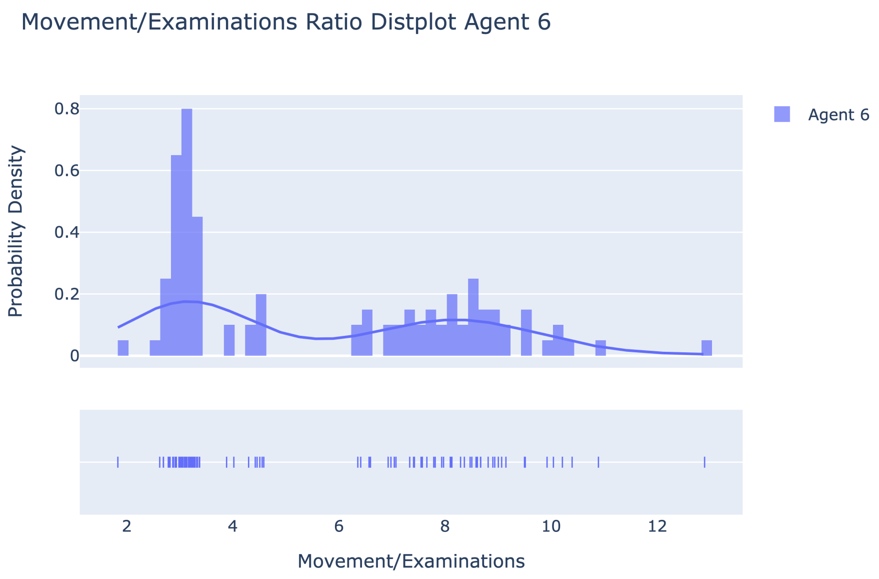


Figure 3: Histogram

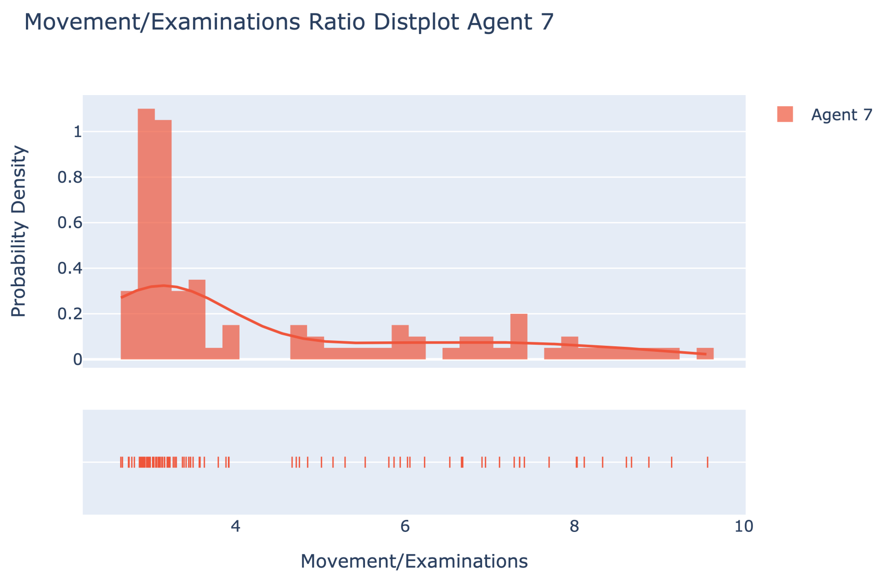


Figure 4: Histogram

From the graphs above and the table in the beginning of this question, we see that the movement/examination ratios for Agent 7 were typically more smaller than the movement/examination ratios of Agent 6 (on average, Agent 7 had a ratio of 4.48, compared to Agent 6 with an average ratio of 5.64). This may likely be due to Agent 7 being able to stop and replan after learning the false negative rate of its current cell which could've changed the cell with the highest probability of finding the target. As a result, we end up with less movements and more examinations compared to Agent 6. The box plot below also shows that the spread of the ratios for 7 is generally less than the spread for Agent 6.

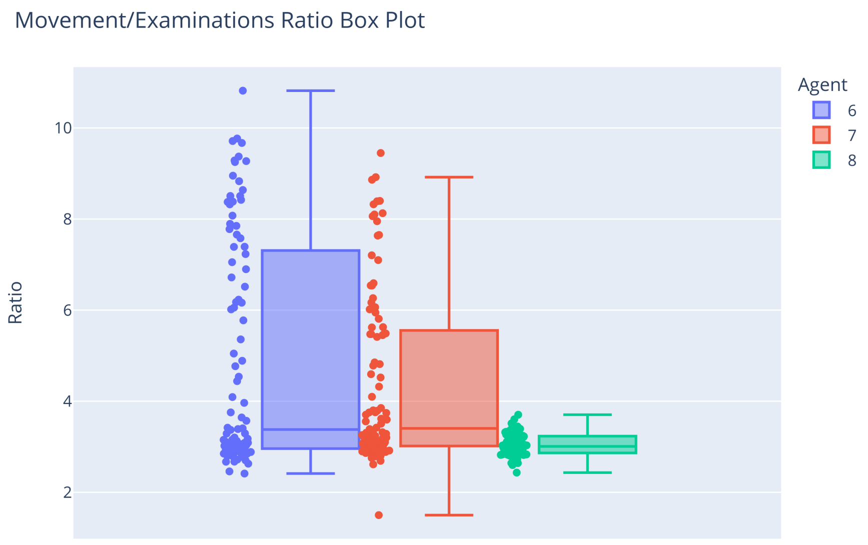


Figure 5: Box Plot

Overall, we can observe from this data that Agent 7 is superior over Agent 6.

Describe your algorithm, be explicit as to what decisions it is making, how, and why. How does the belief state $(P_{i,j}(t))$ enter into the decision making? Do you need to calculate anything new that you didn't already have available?

Like the initial two agents, agent 8 roughly follows the same process. It first makes a guess as to where the target is on the board and uses repeated A* to work its way towards it.

We first decided to start building Agent 8 from Agent 7 because we noticed that our Agent 7 was able to find the target in less actions than Agent 6 on average.

Once it reaches the first guess, it attempts to look for the target. Each square in the grid has a varying false negative rate depending on the type of terrain. If the target does not exist in the first guess or if the search returns a false negative, we make decisions that differs from Agent 7.

First, we calculate a *utility* value to determine which cell to examine next. This utility value is determined using the probability of finding the target and the manhattan distance from the target. As the probability of finding the target increases, our utility value will increase. However, as the manhattan distance from our current cell increases, our utility value for that cell will decrease. We chose to use this utility value as we don't want to disregard promising cells that are close by just because they don't happen to have the highest probability of finding the target. This way, we can examine closer cells that also have a reasonably high probability of finding the target. This should lower our number of movements as we may find the target in these closer cells which lessens the number of movements incase we go to farther cells and have to come back after examining those farther cells. We use the following formula to determine the utility value of a cell: - $value_{i,j} = p(findingin(i,j)) - dist(agent,(i,j)) * (\frac{1}{dim^2}) * 0.01$

We first normalize the distance value by dividing it by the total size of the grid. This way, we won't severely limit the distance we're willing to travel to a cell with the highest probability of finding the target. For example, in larger grid sizes, the manhattan distance to a promising cell will be large and could be bad for the utility value if we don't normalize the distance. And thus, we divide the manhattan distance by the size of the grid to prevent this issue. We then multiply the distance with an additional 0.01 to further avoid that issue of large distances being too punishing. Now that we determined the utility value of the cells, we pick the cell with the greatest utility value.

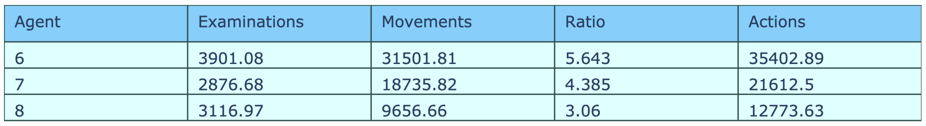
Next, in the event that there is a tie for the highest utility value, we look at the surroundings of all of the tied cells. We do this by summing the probability of finding the target in the neighbors of the tied cells. Whichever cell has the highest sum is then chosen as the next cell to examine. We do this in order to go to an area with high probabilities of finding the cell as this should increase our chances of finding the target without having to move much more incase the cell we examined failed.

Lastly, during execution of our path to the cell with the highest utility value, we also examine cells along the path that also have a relatively high probability of finding the target. We determine this by seeing if that cell has a finding probability that is at least 0.8 * the finding probability of the cell with the highest utility value. This way, we can check reasonable cells along the way without having to perform A* and execute a path towards that cell later on. We chose the value 0.8 as we want to be open to more cells along the path; however, we don't want to examine too many cells along the path if their finding probability isn't reasonably high compared to the cell with the highest utility.

The belief state ($P_{i,j}(t)$) enters into our decision making when we are determining the utility value of a cell, finding cells in areas with higher probabilities, and when deciding to examine cells along the path. This is because we rely on the probability of finding the cell to make these decisions as described above.

We don't need to calculate anything new that isn't already available to us as we use the same probabilities and manhattan distances that we used in Agent 6 and 7. All we do in Agent 8 is use these values in different ways to calculate the utility value of examining a cell to see if that cell is reasonable to examine.

Implement Agent 8, run it sufficiently many times to give a valid comparison to Agents 6 and 7, and verify that Agent 8 is superior.



Agent	Examinations	Movements	Ratio	Actions
6	3901.08	31501.81	5.643	35402.89
7	2876.68	18735.82	4.385	21612.5
8	3116.97	9656.66	3.06	12773.63

Figure 6: Averages Table

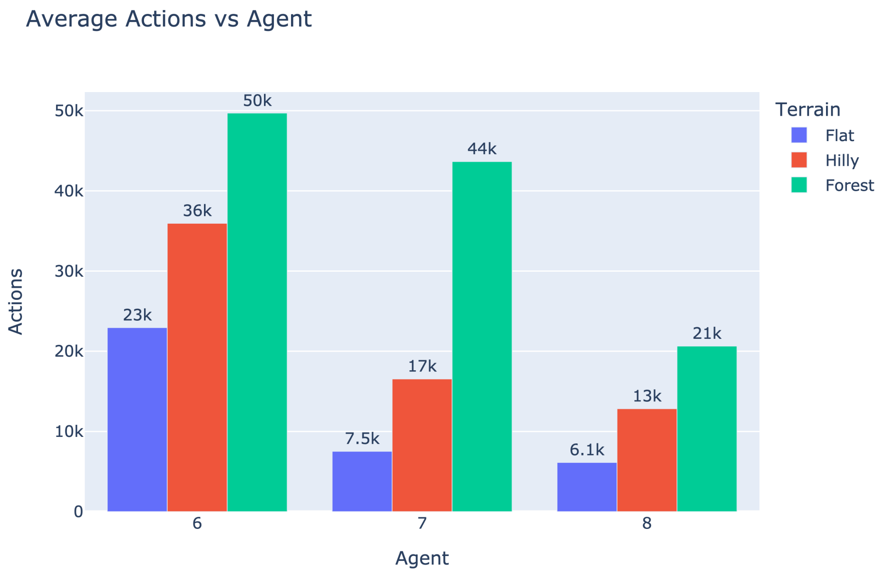


Figure 7: Bar Chart

From the graphs, we can observe that as the false negative rates of the terrains increases, the amount of actions taken by Agent 8 also increases. This makes sense as a higher false negative rate means the chance of failing an examination at the target cell increases, which leads to more actions needed to eventually reexamine the target cell.

However, we also observe the Agent 8 is still able to outperform both Agents 6 and 7 as Agent 8 took fewer actions to find the target. We see this trend among all the terrain types for the target. This is because Agent 8 fixes some of the weaknesses found in Agents 6 and 7.

The weakness Agents 6 and 7 suffers from is that they ignore close cells that also have reasonably high probabilities. By doing this, these Agents will end up having to travel back and forth to examine cells which increases number of movements. Agent 8 fixes this using the various strategies described earlier. By considering closer cells with reasonably high probabilities using utility values, Agent 8 was able to cut down on the number of movements by finding the target earlier.

We also observe that the movement/examination ratio is smaller for Agent 8. This makes sense as we sacrifice a lot of movement for more examinations in our strategies. By taking the chance to examine closer cells with reasonably high probabilities, we increase the number of examinations, but significantly decrease the number of movements. This allows for a smaller ratio in Agent 8 compared to Agents 6 and 7.

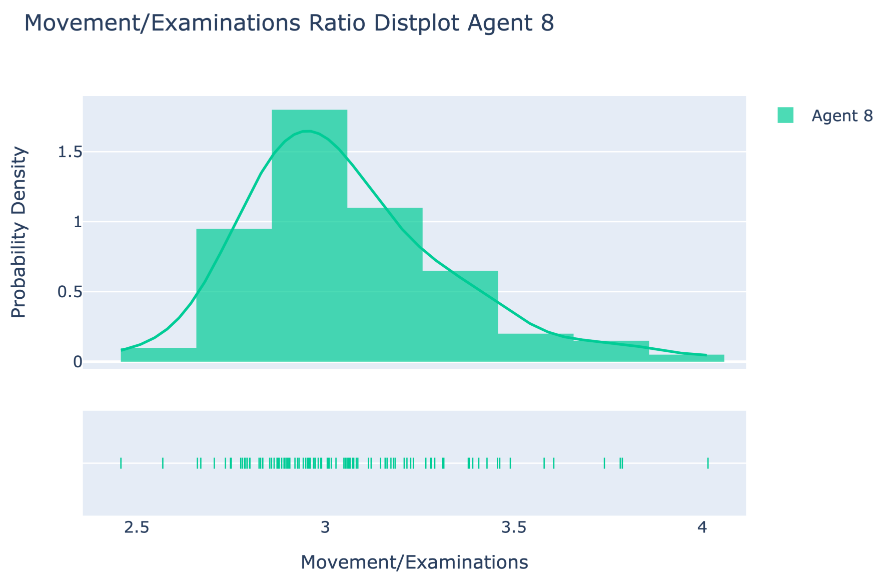


Figure 8: Histogram

The above graph also shows that the distribution of the ratios for Agent 8 is also more compact and less spread apart compared to Agents 6 and 7.

Overall, our data supports that Agent 8 is superior to Agents 6 and 7.

How could you improve Agent 8 even further? Be explicit as to what you could do, how, and what you would need.

One way we could improve Agent 8 further is by adjusting the way we plan our paths. Rather than planning a path towards one cell with the highest utility value, we could have our planning phase plan an optimized path to visit the 5 cells with the highest utility value. By planning a path to one cell at a time, we may not be able to visit the top 5 cells with the highest utility value in the most optimized path. For example, traversing to the second highest utility value cell then to the first may require less movements than traversing to the highest utility value cell then to the second.

To do this, we would first need to keep track of the five cells with the highest utility values. This could be done by keeping a list that keeps track of these top five values as we update probabilities. We would then use our A* planning method to try out the different orderings of these five cells to see which sequences of paths sums up to the shortest among those orderings. We can then traverse the shortest path among the different orderings of these top 5 cells. This should lower the number of movements we make with the agent while also examining the top cells which may likely contain the target.

Appendix

Agent 6

```
from heuristics import manhattan
from gridworld import Gridworld
from probability_queue import Probability_Queue
from probability_node import Probability_Node
from random import random, choice
import heapq

class Agent_6:

  def __init__(self, dim, start):
    self.dim = dim
    self.discovered_grid = Gridworld(dim)
    self.cells = {}
    self.belief_state = []
    # Initialize all cell info in belief state
    for i in range(self.dim):
      for j in range(self.dim):
        prob_node = Probability_Node(1/(self.dim**2), 1/(self.dim**2), (i,j))
        self.belief_state.append(prob_node)
        self.cells[(i,j)] = prob_node
    self.max_cell = self.cells[start]

  def execute_path(self, path, complete_grid, guess, target):
    movements = 0
    examinations = 0
    for node in path:
      curr = node.curr_block

      # check if path is blocked
      if complete_grid.gridworld[curr[0]][curr[1]] == 1:
        # update our knowledge of blocked nodes
        self.discovered_grid.update_grid_obstacle(curr, 1)
        # update the belief state after learning about this blocked cell
        self.update_belief_block(curr, node.parent_block.curr_block)
        return node.parent_block, movements, examinations, False

      # Update discovered grid with the terrain type
      self.discovered_grid.update_grid_obstacle(curr, complete_grid.gridworld[curr[0]][curr
      # Update cell's false negative rate after observing its terrain type
      self.update_false_negative_rate(curr)
      # Increment number of actions taken because of movement
      movements += 1

      # Check if we reached target cell
      if curr == guess:
        # Increment number of actions taken because of examination
        examinations += 1
        # Examine the cell to search for target
        if self.examine_cell(curr, target):
          return path[-1], movements, examinations, True
    return path[-1], movements, examinations, False

  # Return max cell
  def get_max_cell(self, curr):
    return self.max_cell.coord

  # Update given cell's false negative rate based off its terrain type
  def update_false_negative_rate(self, coord):
    # Observe the terrain at this cell and update its false negative rate appropriately
    if self.discovered_grid.gridworld[coord[0]][coord[1]] == 2:
      self.cells[coord].false_negative_rate = 0.2
    elif self.discovered_grid.gridworld[coord[0]][coord[1]] == 3:
      self.cells[coord].false_negative_rate = 0.5
    elif self.discovered_grid.gridworld[coord[0]][coord[1]] == 4:
      self.cells[coord].false_negative_rate = 0.8

  # Examine this cell to see if target is here
  def examine_cell(self, coord, target):
    # Check if target is even contained in coord
    if coord != target:
      self.update_belief_failed_examination(coord)
      return False
    # If we are in the cell with the target, simulate a search
    prob_node = self.cells[coord]
    if random() > prob_node.false_negative_rate:
      return True
    else:
      self.update_belief_failed_examination(coord)
      return False

  # Update probabilities of all cells after learning about the blocked cell
  def update_belief_block(self, block_coord, last_coord):
    # Probabilty that the blocked cell contained the target
    block_probability = self.cells[block_coord].target_probability
    # Reset max_cell
    self.max_cell = self.belief_state[0]
    # list of max cells to break a tie
    max_cells = []

    # Update the beliefs of each cell
    for prob_node in self.belief_state:
      if not prob_node.coord == block_coord:
        # Update probabilities of all cells except the blocked cell
        prob_node.target_probability = prob_node.target_probability / (1 - block_probability
        prob_node.priority_probability = prob_node.target_probability
      else:
        # Update probabilities of the blocked cell
        prob_node.target_probability = 0
        prob_node.priority_probability = 0

      # update max cell if necessary
      if prob_node.priority_probability > self.max_cell.priority_probability:
        self.max_cell = prob_node
        max_cells = []
      # if probabilities are the same use the distance to break a tie
      elif prob_node.priority_probability == self.max_cell.priority_probability:
        if manhattan(last_coord, self.max_cell.coord) > manhattan(last_coord, prob_node.coor
          max_cells = []
          self.max_cell = prob_node
        # if distances and probabilities are the same use uniform random to break a tie
        elif manhattan(last_coord, self.max_cell.coord) == manhattan(last_coord, prob_node.c
          if not max_cells:
            max_cells.append(self.max_cell)
          max_cells.append(prob_node)

    # Uniform randomly pick a cell
    if max_cells:
      self.max_cell = choice(max_cells)


    # Heapify the queue so the cell with max probability is next in queue
    #heapq.heapify(self.belief_state.queue)
    #print(self.belief_state.queue)

  # Update probabilities of all cells if examination fails
  def update_belief_failed_examination(self, coord):
    # Probability that the examined cell contained the target
    examine_probability = self.cells[coord].target_probability
    # False negative rate of the examined cell
    examine_false_negative_rate = self.cells[coord].false_negative_rate
    # Reset max_cell
    self.max_cell = self.belief_state[0]
    # used to break ties
    max_cells = []

    # Update the beliefs of each cell
    for prob_node in self.belief_state:
      if not prob_node.coord == coord:
        # Update probabilities of all cells except the examined cell
        prob_node.target_probability = prob_node.target_probability / (examine_probability *
        prob_node.priority_probability = prob_node.target_probability
      else:
        # Update probabilities of the examined cell
        prob_node.target_probability = (examine_probability * examine_false_negative_rate) /
        prob_node.priority_probability = prob_node.target_probability

      # update max cell if necessary
      if prob_node.priority_probability > self.max_cell.priority_probability:
        self.max_cell = prob_node
        max_cells = []
      # if probabilities are the same use the distance to break a tie
      elif prob_node.priority_probability == self.max_cell.priority_probability:
        if manhattan(coord, self.max_cell.coord) > manhattan(coord, prob_node.coord):
          max_cells = []
          self.max_cell = prob_node
        # if distances and probabilities are the same use uniform random to break a tie
        elif manhattan(coord, self.max_cell.coord) == manhattan(coord, prob_node.coord):
          if not max_cells:
            max_cells.append(self.max_cell)
          max_cells.append(prob_node)

    # Uniform randomly pick a cell
    if max_cells:
      self.max_cell = choice(max_cells)

    # Heapify the queue so the cell with max probability is next in queue
    #heapq.heapify(self.belief_state.queue)
    #print(self.belief_state.queue)
```

Agent 7

```
from heuristics import manhattan
from gridworld import Gridworld
from probability_queue import Probability_Queue
from probability_node import Probability_Node
from random import random, choice
import heapq

class Agent_7:

  def __init__(self, dim, start):
    self.dim = dim
    self.discovered_grid = Gridworld(dim)
    self.cells = {}
    self.belief_state = []
    # Initialize all cell info in belief state
    for i in range(self.dim):
      for j in range(self.dim):
        prob_node = Probability_Node(1/(self.dim**2) * 0.35, 1/(self.dim**2), (i,j))
        self.belief_state.append(prob_node)
        self.cells[(i,j)] = prob_node
    self.max_cell = self.cells[start]

  def execute_path(self, path, complete_grid, guess, target):
    movements = 0
    examinations = 0
    for node in path:
      curr = node.curr_block

      # check if path is blocked
      if complete_grid.gridworld[curr[0]][curr[1]] == 1:
        # update our knowledge of blocked nodes
        self.discovered_grid.update_grid_obstacle(curr, 1)
        # update the belief state after learning about this blocked cell
        self.update_belief_block(curr, node.parent_block.curr_block)
        return node.parent_block, movements, examinations, False

      # Update discovered grid with the terrain type
      self.discovered_grid.update_grid_obstacle(curr, complete_grid.gridworld[curr[0]][curr
      # Update cell's false negative rate after observing its terrain type
      self.update_false_negative_rate(curr)
      # Increment number of actions taken because of movement
      movements += 1

      # Update probability of this cell after discovering terrain
      self.update_priority_probability(curr)

      # Check if we reached target cell
      if curr == guess:
        # Increment number of actions taken because of examination
        examinations += 1
        # Examine the cell to search for target
        if self.examine_cell(curr, target):
          return path[-1], movements, examinations, True

      # Check if cell with highest probability changed
      if self.cells[curr].priority_probability >= self.max_cell.priority_probability:
        self.max_cell = self.cells[curr]
        return node, movements, examinations, False

    return path[-1], movements, examinations, False

  # Return max cell
  def get_max_cell(self, curr):
    return self.max_cell.coord

  # Update given cell's false negative rate based off its terrain type
  def update_false_negative_rate(self, coord):
    # Observe the terrain at this cell and update its false negative rate appropriately
    if self.discovered_grid.gridworld[coord[0]][coord[1]] == 2:
      self.cells[coord].false_negative_rate = 0.2
    elif self.discovered_grid.gridworld[coord[0]][coord[1]] == 3:
      self.cells[coord].false_negative_rate = 0.5
    elif self.discovered_grid.gridworld[coord[0]][coord[1]] == 4:
      self.cells[coord].false_negative_rate = 0.8

  # Updates probability of cell based off its false negative rate
  def update_priority_probability(self, coord):
    self.cells[coord].priority_probability = self.cells[coord].target_probability * (1 - sel

  # Examine this cell to see if target is here
  def examine_cell(self, coord, target):
    # Check if target is even contained in coord
    if coord != target:
      self.update_belief_failed_examination(coord)
      return False
    # If we are in the cell with the target, simulate a search
    prob_node = self.cells[coord]
    if random() > prob_node.false_negative_rate:
      return True
    else:
      self.update_belief_failed_examination(coord)
      return False

  # Update probabilities of all cells after learning about the blocked cell
  def update_belief_block(self, block_coord, last_coord):
    # Probabilty that the blocked cell contained the target
    block_probability = self.cells[block_coord].target_probability
    # Reset max_cell
    self.max_cell = self.belief_state[0]
    # list of max cells to break a tie
    max_cells = []

    # Update the beliefs of each cell
    for prob_node in self.belief_state:
      if not prob_node.coord == block_coord:
        # Update probabilities of all cells except the blocked cell
        prob_node.target_probability = prob_node.target_probability / (1 - block_probability
        prob_node.priority_probability = prob_node.target_probability * (1 - prob_node.false
      else:
        # Update probabilities of the blocked cell
        prob_node.target_probability = 0
        prob_node.priority_probability = 0

      # update max cell if necessary
      if prob_node.priority_probability > self.max_cell.priority_probability:
        self.max_cell = prob_node
        max_cells = []
      # if probabilities are the same use the distance to break a tie
      elif prob_node.priority_probability == self.max_cell.priority_probability:
        if manhattan(last_coord, self.max_cell.coord) > manhattan(last_coord, prob_node.coor
          max_cells = []
          self.max_cell = prob_node
        # if distances and probabilities are the same use uniform random to break a tie
        elif manhattan(last_coord, self.max_cell.coord) == manhattan(last_coord, prob_node.c
          if not max_cells:
            max_cells.append(self.max_cell)
          max_cells.append(prob_node)

    # Uniform randomly pick a cell
    if max_cells:
      self.max_cell = choice(max_cells)


    # Heapify the queue so the cell with max probability is next in queue
    #heapq.heapify(self.belief_state.queue)
    #print(self.belief_state.queue)

  # Update probabilities of all cells if examination fails
  def update_belief_failed_examination(self, coord):
    # Probability that the examined cell contained the target
    examine_probability = self.cells[coord].target_probability
    # False negative rate of the examined cell
    examine_false_negative_rate = self.cells[coord].false_negative_rate
    # Reset max_cell
    self.max_cell = self.belief_state[0]
    # used to break ties
    max_cells = []

    # Update the beliefs of each cell
    for prob_node in self.belief_state:
      if not prob_node.coord == coord:
        # Update probabilities of all cells except the examined cell
        prob_node.target_probability = prob_node.target_probability / (examine_probability *
        prob_node.priority_probability = prob_node.target_probability * (1 - prob_node.false
      else:
        # Update probabilities of the examined cell
        prob_node.target_probability = (examine_probability * examine_false_negative_rate) /
        prob_node.priority_probability = prob_node.target_probability * (1 - prob_node.false

      # update max cell if necessary
      if prob_node.priority_probability > self.max_cell.priority_probability:
        self.max_cell = prob_node
        max_cells = []
      # if probabilities are the same use the distance to break a tie
      elif prob_node.priority_probability == self.max_cell.priority_probability:
        if manhattan(coord, self.max_cell.coord) > manhattan(coord, prob_node.coord):
          max_cells = []
          self.max_cell = prob_node
        # if distances and probabilities are the same use uniform random to break a tie
        elif manhattan(coord, self.max_cell.coord) == manhattan(coord, prob_node.coord):
          if not max_cells:
            max_cells.append(self.max_cell)
          max_cells.append(prob_node)

    # Uniform randomly pick a cell
    if max_cells:
      self.max_cell = choice(max_cells)

    # Heapify the queue so the cell with max probability is next in queue
    #heapq.heapify(self.belief_state.queue)
    #print(self.belief_state.queue)
```

Agent 8

```
from heuristics import manhattan
from gridworld import Gridworld
from probability_queue import Probability_Queue
from probability_node import Probability_Node
from random import random, choice
import heapq

class Agent_8:

  def __init__(self, dim, start):
    self.dim = dim
    self.discovered_grid = Gridworld(dim)
    self.cells = {}
    self.belief_state = []
    # Initialize all cell info in belief state
    for i in range(self.dim):
      for j in range(self.dim):
        prob_node = Probability_Node(1/(self.dim**2) * 0.35, 1/(self.dim**2), (i,j))
        self.belief_state.append(prob_node)
        self.cells[(i,j)] = prob_node
    self.max_cell = self.cells[start]

  def execute_path(self, path, complete_grid, guess, target):
    movements = 0
    examinations = 0
    for node in path:
      curr = node.curr_block

      # check if path is blocked
      if complete_grid.gridworld[curr[0]][curr[1]] == 1:
        # update our knowledge of blocked nodes
        self.discovered_grid.update_grid_obstacle(curr, 1)
        # update the belief state after learning about this blocked cell
        self.update_belief_block(curr, node.parent_block.curr_block)
        return node.parent_block, movements, examinations, False

      # Update discovered grid with the terrain type
      self.discovered_grid.update_grid_obstacle(curr, complete_grid.gridworld[curr[0]][curr
      # Update cell's false negative rate after observing its terrain type
      self.update_false_negative_rate(curr)
      # Increment number of actions taken because of movement
      movements += 1

      # Update probability of this cell after discovering terrain
      self.update_priority_probability(curr)

      # Check if we reached target cell
      if self.cells[curr].priority_probability >= 0.8 * self.max_cell.priority_probability:
        # Increment number of actions taken because of examination
        examinations += 1
        # Examine the cell to search for target
        if self.examine_cell(curr, target):
          return path[-1], movements, examinations, True

      # Check if cell with highest probability changed
      if self.cells[curr].priority_probability >= self.max_cell.priority_probability:
        self.max_cell = self.cells[curr]
        return node, movements, examinations, False

    return path[-1], movements, examinations, False

  # Return max cell
  def get_max_cell(self, curr):
    return self.max_cell.coord

  # Update given cell's false negative rate based off its terrain type
  def update_false_negative_rate(self, coord):
    # Observe the terrain at this cell and update its false negative rate appropriately
    if self.discovered_grid.gridworld[coord[0]][coord[1]] == 2:
      self.cells[coord].false_negative_rate = 0.2
    elif self.discovered_grid.gridworld[coord[0]][coord[1]] == 3:
      self.cells[coord].false_negative_rate = 0.5
    elif self.discovered_grid.gridworld[coord[0]][coord[1]] == 4:
      self.cells[coord].false_negative_rate = 0.8

  # Updates probability of cell based off its false negative rate
  def update_priority_probability(self, coord):
    self.cells[coord].priority_probability = self.cells[coord].target_probability * (1 - sel

  # Examine this cell to see if target is here
  def examine_cell(self, coord, target):
    # Check if target is even contained in coord
    if coord != target:
      self.update_belief_failed_examination(coord)
      return False
    # If we are in the cell with the target, simulate a search
    prob_node = self.cells[coord]
    if random() > prob_node.false_negative_rate:
      return True
    else:
      self.update_belief_failed_examination(coord)
      return False

  # Update probabilities of all cells after learning about the blocked cell
  def update_belief_block(self, block_coord, last_coord):
    # Probabilty that the blocked cell contained the target
    block_probability = self.cells[block_coord].target_probability
    # Reset max_cell
    self.max_cell = self.belief_state[0]
    # list of max cells to break a tie
    max_cells = []

    # Update the beliefs of each cell
    for prob_node in self.belief_state:
      if not prob_node.coord == block_coord:
        # Update probabilities of all cells except the blocked cell
        prob_node.target_probability = prob_node.target_probability / (1 - block_probabilit
        prob_node.priority_probability = prob_node.target_probability * (1 - prob_node.false
        prob_node.value = prob_node.priority_probability - manhattan(last_coord, prob_node.
      else:
        # Update probabilities of the blocked cell
        prob_node.target_probability = 0
        prob_node.priority_probability = 0
        prob_node.value = prob_node.priority_probability - manhattan(last_coord, prob_node.

      # update max cell if necessary
      if prob_node.priority_probability != 0 and prob_node.value > self.max_cell.value:
        self.max_cell = prob_node
        max_cells = []
      # if probabilities are the same use the distance to break a tie
      elif prob_node.value == self.max_cell.value:
        if not max_cells:
          max_cells.append(self.max_cell)
        max_cells.append(prob_node)

    # Don't randomly pick a cell, pick the one with the highest surrounding prob
    if max_cells:
      # list used to check the neighbors in four directions
      to_check = [(0, 1), (0, -1), (-1, 0), (1, 0)]
      # keep track of the maximum probability
```

```
      max_prob = 0
      # keep track of the cell with max prob neighbors
      temp_max_cell = choice(max_cells)

      for c in max_cells:
        # current probability counter
        curr_prob = 0
        # loop through and check the neighbors
        for n in to_check:
          # the cordinates of the neighbor
          curr_neighbor = (c.coord[0] + n[0], c.coord[1] + n[1])
          # check bounds
          if curr_neighbor[0] >= 0 and curr_neighbor[0] < self.dim and curr_neighbor[1] >= 0
            # check that the neighbor is not a block
            if self.discovered_grid.gridworld[curr_neighbor[0]][curr_neighbor[1]] not in (1,
              neighbor = self.cells[curr_neighbor]
              curr_prob += neighbor.target_probability
        if curr_prob > max_prob:
          max_prob = curr_prob
          temp_max_cell = c
      self.max_cell = temp_max_cell




    # Heapify the queue so the cell with max probability is next in queue
    #heapq.heapify(self.belief_state.queue)
    #print(self.belief_state.queue)

  # Update probabilities of all cells if examination fails
  def update_belief_failed_examination(self, coord):
    # Probability that the examined cell contained the target
    examine_probability = self.cells[coord].target_probability
    # False negative rate of the examined cell
    examine_false_negative_rate = self.cells[coord].false_negative_rate
    # Reset max_cell
    self.max_cell = self.belief_state[0]
    # used to break ties
    max_cells = []

    # Update the beliefs of each cell
    for prob_node in self.belief_state:
      if not prob_node.coord == coord:
        # Update probabilities of all cells except the examined cell
        prob_node.target_probability = prob_node.target_probability / (examine_probability *
        prob_node.priority_probability = prob_node.target_probability * (1 - prob_node.false
        prob_node.value = prob_node.priority_probability - manhattan(coord, prob_node.coord)
      else:
        # Update probabilities of the examined cell
        prob_node.target_probability = (examine_probability * examine_false_negative_rate) /
        prob_node.priority_probability = prob_node.target_probability * (1 - prob_node.false
        prob_node.value = prob_node.priority_probability - manhattan(coord, prob_node.coord)

      # update max cell if necessary
      if prob_node.priority_probability != 0 and prob_node.value > self.max_cell.value:
        self.max_cell = prob_node
        max_cells = []
      # if probabilities are the same use the distance to break a tie
      elif prob_node.value == self.max_cell.value:
        if not max_cells:
          max_cells.append(self.max_cell)
        max_cells.append(prob_node)

    # Don't randomly pick a cell, pick the one with the highest surrounding prob
    if max_cells:
      # list used to check the neighbors in four directions
      to_check = [(0, 1), (0, -1), (-1, 0), (1, 0)]
      # keep track of the maximum probability
      max_prob = 0
      # keep track of the cell with max prob neighbors
      temp_max_cell = choice(max_cells)

      for c in max_cells:
        # current probability counter
        curr_prob = 0
        # loop through and check the neighbors
        for n in to_check:
          # the cordinates of the neighbor
          curr_neighbor = (c.coord[0] + n[0], c.coord[1] + n[1])
          # check bounds
          if curr_neighbor[0] >= 0 and curr_neighbor[0] < self.dim and curr_neighbor[1] >= 0
            # check that the neighbor is not a block
            if self.discovered_grid.gridworld[curr_neighbor[0]][curr_neighbor[1]] not in (1,
              neighbor = self.cells[curr_neighbor]
              curr_prob += neighbor.target_probability
        if curr_prob > max_prob:
          max_prob = curr_prob
          temp_max_cell = c
      self.max_cell = temp_max_cell

    # Heapify the queue so the cell with max probability is next in queue
```

```
    #heapq.heapify(self.belief_state.queue)
    #print(self.belief_state.queue)
```

Probability Node

```
class Probability_Node:

    # priority_probability: probability that will be used by probability_queue for ordering
    #                        (prob of cell containing target for agent 6, prob of finding ta
    # target_probability: probabilty of cell containing target
    # distance: Distance from cell to target
    # coord: The coordinates of this cell
    def __init__(self, priority_probability, target_probability, coord):
        self.priority_probability = priority_probability
        self.target_probability = target_probability
        self.coord = coord
        self.value = 0
        self.false_negative_rate = 0.65
```

Main Function

```
import argparse
from time import sleep, time
from agent_6 import Agent_6
from agent_7 import Agent_7
from agent_8 import Agent_8
from gridworld import Gridworld
from heuristics import manhattan
from a_star import path_planner
import json
from pprint import pprint
from random import choices, randint, random


"""
  Creates a gridworld and carrys out repeated A* based on the agent
  @param dim: dimension of the grid
  @param prob: probability of having a blocker
  @param agent: the type of visibility we have
  @param complete_grid: optional supplied grid instead of creating one
"""
def solver(dim, prob, complete_grid=None):

  pro_start_time = time()
  agent_6_actions = 0
  agent_7_actions = 0
```

```
agent_8_actions = 0
agent_6_list = []
agent_7_list = []
agent_8_list = []
runs = 1
for i in range(runs):

  # create a start and end position randomly
  start = (randint(0, dim-1), randint(0, dim-1))
  target = (randint(0, dim-1), randint(0, dim-1))
  guess = start

  # json output
  data = {}

  # create a gridworld
  # print("Start: " + str(start))
  # print("Target: " + str(target))
  # print("Guess: " + str(guess))
  # keep generating a new grid until we get a solvable one
  complete_grid = Gridworld(dim, start, target, prob, False)
  while not verify_solvability(dim, start, target, complete_grid):
    # keep generating a new grid until we get a solvable one
    complete_grid = Gridworld(dim, start, target, prob, False)
  # complete_grid.print()
  # print()

  # create agents
  agents = [Agent_6(dim, start), Agent_7(dim, start), Agent_8(dim, start)]
  # agents = [Agent_8_ac(dim, start)]
  agent_counter = 5

  for agent_object in agents:
    agent_counter += 1
    agent_start = start
    agent_guess = guess
    # total number of movements and examinations taken
    total_movements = 0
    total_examinations = 0
    # total number of cells processed
    total_cells_processed = 0
    # final path which points to last node
    final_path = None
    # number of times A* was repeated
    retries = 0
    # status of completion
```

```
complete_status = False

# perform repeated A* with the agent
starting_time = time()
# start planning a path from the starting block
new_path, cells_processed = path_planner(
    agent_start, final_path, agent_guess, agent_object.discovered_grid, dim, manhattan
)
total_cells_processed += cells_processed
# while A* finds a new path
while len(new_path) > 0:
    retries += 1
    # execute the path
    last_node, movements, examinations, found_target = agent_object.execute_path(new_p
    total_movements += movements
    total_examinations += examinations
    final_path = last_node
    # get the last unblocked block
    last_unblock_node = None
    if last_node:
        agent_start = last_node.curr_block
        last_unblock_node = last_node.parent_block
    # check if target was found
    if found_target:
        complete_status = True
        break
    # Update guess cell to be next cell with highest probability
    agent_guess = agent_object.get_max_cell(agent_start)

    # print("Run: " + str(i) + " Agent: " + str(agent_counter) + " New Start: " + str

    # create a new path from the last unblocked node
    new_path, cells_processed = path_planner(
        agent_start,
        last_unblock_node,
        agent_guess,
        agent_object.discovered_grid,
        dim,
        manhattan,
    )
    total_cells_processed += cells_processed
    # If there is no path to the guess, treat it as blocked and keep targeting the nex
    while not new_path:
      # Treat guess as blocked and update beliefs
      agent_object.update_belief_block(agent_guess, agent_start)
      # Make new guess using the cell with highest probability
```

```
            agent_guess = agent_object.get_max_cell(agent_start)
            retries += 1
            # Find a path to this new guess cell
            new_path, cells_processed = path_planner(
              agent_start,
              last_unblock_node,
              agent_guess,
              agent_object.discovered_grid,
              dim,
              manhattan
            )
            total_cells_processed += cells_processed

      completion_time = time() - starting_time
      data["Agent {}".format(agent_counter)] = {"processed": total_cells_processed, "retries

      # print("Agent %s Completed in %s seconds" % (agent_counter, completion_time))
      # print("Agent %s Processed %s cells" % (agent_counter, total_cells_processed))
      # print("Agent %s Took %s actions" % (agent_counter, total_actions))
      # print("Target found: " + str(target) + " with type " + str(complete_grid.gridworld[
      # if agent_counter == 6:
      #   agent_6_actions += total_actions
      #   agent_6_list.append(total_actions)
      # elif agent_counter == 7:
      #   agent_7_actions += total_actions
      #   agent_7_list.append(total_actions)
      # elif agent_counter == 8:
      #   agent_8_actions += total_actions
      #   agent_8_list.append(total_actions)

    # pprint(data)
  print(json.dumps(data))
  # print("List of Agent 6 Actions: " + str(agent_6_list))
  # print("List of Agent 7 Actions: " + str(agent_7_list))
  # print("List of Agent 8 Actions: " + str(agent_8_list))
  # print("Agent 6 Average Actions: " + str(agent_6_actions/runs))
  # print("Agent 7 Average Actions: " + str(agent_7_actions/runs))
  # print("Agent 8 Average Actions: " + str(agent_8_actions/runs))
  # print("Took %s seconds" % (time() - pro_start_time))


def verify_solvability(dim, start, target, complete_grid):
    # start planning a path from the starting block
    new_path, cells_processed = path_planner(start, None, target, complete_grid, dim, manhat

    # Check if a path was found
    if not new_path:
      return False

    return True

def main():
    p = argparse.ArgumentParser()
    p.add_argument(
      "-d", "--dimension", type=int, default=25, help="dimension of gridworld"
    )
    p.add_argument(
      "-p",
      "--probability",
      type=float,
      default=0.30,
      help="probability of a blocked square",
    )

    # parse arguments and create the gridworld
    args = p.parse_args()

    # call the solver method with the args
    solver(args.dimension, args.probability)

if __name__ == "__main__":
    main()
```