Project 2

CS520 - Intro to Artificial Intelligence

Muzzammil Shahe Ansar (fm501)
Harshankar Reddy Gudur (hsg64)
Vishal Singh (vs872)

1

Bot 1 and Bot 2

The design and implementation of Bot 1 and Bot 2 are stated here. In this situation, we have one crew member that needs rescuing, and one alien in play.

1.1 Bot 1

Bot 1 is aware of two things at the start:

- It knows that the crew member is equally likely to be in any other open cell other than the bot's initial cell.
- It also knows that the alien is equally likely to be in any other open cell outside the bot's detection square.

The bot has access to two kinds of sensor data at every single timestep.

- Alien sensor: For a given k, the bot can sense whether there is an alien in the (2k + 1) × (2k + 1) square centered at its location.
- Crew sensor: For a given $\alpha > 0$, if there is a crew member hidden 'd' distance away (Manhattan distance) from the bot, the bot receives a beep with probability

$$e^{-\alpha(d-1)} \tag{1.1}$$

At every point in time, we make updates to the crew and alien belief system based on the sensor data we receive. The bots path planning (utilizing Breadth First Search) is designed in a way to move towards the cell which has the highest crew belief (the cell which has the highest probability of containing a crew member) and by sticking to the cells which most definitely do not contain an alien.

1.1.1 Mathematical Modeling

1.1.1.1 Probability

The situation is this: Our bot is in a 35 x 35 grid (or "ship") with no idea of where the crew member is, except for the one hint it receives: a beep. With some probability, it will hear a beep from the crew member. Using this information, the bot has to zone in on the location of the crew member ultimately locating its precise coordinates. For now let's ignore the fact that it has to do all this while also evading an alien in the ship, and focus more on this beep. Considering the incredibly low number of

aliens, this is not a bad idea.
It's given to us that the bot receives a beep from the crew member with a probability:

$$P(Beep) = e^{-\alpha(d-1)}$$

where,
$d \rightarrow$ the Manhattan Distance from the bots' current position to the crew member
$\alpha \rightarrow$ some positive, small, non-zero constant
The bot uses this information to locate the crew member. But how does the bot utilize this to adjust its knowledge about the location of the crew member?
While the beep itself doesn't contain any detail about location, looking at the formula we can see that the probability of receiving a beep depends on the distance from the crew member.
Suppose the crew member is in the neighboring cells of the bot, our value of d would equal to 1 making our P(receiving a beep) also equal to 1. The probability decreases as we go further away (our value of d increases). So hearing a beep now indicates that we are probably close to the crew member. The more the frequency of the beeps increase, the more the chance of the crew member being close by.
Going by this intuition, we use Bayes Theorem to design the probabilistic model for our bot.

1.1.1.2 Bayes' Theorem

Bayes' Theorem describes the probability of an event A given that another event B has already taken place. This probability is denoted as $P(A|B)$ (probability of A given B). Mathematically put-

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

$P(A|B) \rightarrow$ probability of event A occurring given event B has occurred
$P(B|A) \rightarrow$ probability of event B occurring given event A has occurred
$P(A) \rightarrow$ probability of event A occurring
$P(B) \rightarrow$ probability of event B occurring

Here, $P(A|B)$ is called the posterior, and $P(A)$ is called the prior. Bayes' Theorem is a powerful tool. It allows us to modify belief with measurements and predictions. Most Bayesian filters are a combination of two things - prediction and measurement. We predict to get a prior. We then measure to get the likelihood. We can now combine both to get a posterior which gives us a much better picture of whatever it is that we are trying to understand. So another way to write Bayes' theorem would be

$$Posterior = \frac{Likelihood \cdot Prior}{Normalization\ Constant}$$

We borrow help from Bayes to allow our bots to approximate the location of its crew members.

1.1.1.3 Crew Member Belief

Bot 1 does not know where the crew members are. It can only track which ones are more likely given the data from the sensor from time $t = t_0$ to $t = t_i$. We can use the above method of Bayesian filtering to incorporate all measurements and predictions into something that eventually will end up pointing us toward the crew member. To do that we implement a simple **discrete Bayesian filter**.
We associate with each cell a belief that the crew member is there. All are initialized with equal beliefs. Every update we try to detect a beep. Depending on if we detect a beep or not, we get the following probability-

$$P(Beep|Crew_x) = e^{-\alpha(d-1)}$$

$$P(\neg Beep|Crew_x) = 1 - P(Beep|Crew_x)$$

$Crew_x$ is notation for crew at some x, which is just the location of the cell.
We can apply Bayes' Theorem to the above to find what we want-

$$P(Crew_x)_{t+1} = P(Crew_x|Beep)_t = \eta P(Beep|Crew_x)_t \cdot P(Crew_x)_t$$

$$P(Crew_x)_{t+1} = P(Crew_x|\neg Beep)_t = \eta P(\neg Beep|Crew_x)_t \cdot P(Crew_x)_t$$

We iterate through every cell, set x to the location of the cell and run either of the above two computations to get our new beliefs. We also make sure to keep track of the cells we have already discovered and set their belief to 0. This is a complete mathematical description of how Bot 1 manages to find the crew member.

1.1.1.4 Alien Belief

Alien Beliefs work similarly to the above. We do not have a closed-form analytical equation but we can use what we know about aliens to model them. We know this about aliens - they choose randomly from one of their neighbors and move there. They will keep on doing this for the entire run. We also have a sensor. This sensor has 100% accuracy but only has a binary output with a fixed range, i.e., if an alien is within a square of a certain size centered at the bot's position, the sensor will output 1. If not, it will output 0. This provides us with some interesting information - we can only tell if the alien is within the square. We do not know where. We, however, do have the two necessary ingredients for a Bayes' filter - a model for prediction, and a sensor for measurement.
Let us start with a description of the model. Let us denote the neighborhood of an alien by $N_A(x)$. If an alien is at x at time t, we can make the following predictions-

1. At time $t + 1$, the alien will move to one of the positions in $N_A(x)_t$, if at least one such position is available
2. At time $t + 1$, the alien belief at x will be 0, assuming $N_A(x)_t$ was not empty

If we were to talk in terms of belief, we can put the above like this-

$$P(Alien_{x_n})_{t+1} = \frac{P(Alien_x)_t}{n(N_A(x)_t)}, x_n \in N_A(x)_t$$

$$P(Alien_x)_{t+1} = 0$$

We call this distribution of belief diffusion. Because the grid is connected, there will always be at least one neighbor. Therefore, we will not consider the case where the alien has 0 neighbors. The above distributes the belief that the alien is at a cell into its neighbors. We also need to keep in mind that other cells exist and they may have their own beliefs distributed, or waiting to be distributed too. So in reality, the above should be thought more of as a superposition of all of them. Setting the belief directly to 0 can be avoided by using a new grid initialized to 0. We will see this in detail in the implementation section.
Now we need to talk about the measurement part. The above gives us our prior. Now that we have our prior, we need to update it with measurements. We can say the following regarding measurements-

1. If the sensor detects an alien, only the beliefs inside of the detection square matter
2. If the sensor does not detect an alien, only the beliefs outside of the detection square matter

There is more to be said regarding the definition of N_A since it gets complicated by the detection square. This will be explained in detail in the implementation section. For now, we use the above to localize the aliens. We can put this in terms of probability-

$$P(Detection|Alien_x) = \begin{cases} 1 & x \in Detection\ Square \\ 0 & x \notin Detection\ Square \end{cases}$$

$$P(\neg Detection|Alien_x) = 1 - P(Detection|Alien_x)$$

The above can be used to find the two equations necessary to update our alien's beliefs-

$$P(Alien_x|Detection)_t = \eta P(Detection|Alien_x)_t \cdot P(Alien_x)_t$$

$$P(Alien_x|\neg Detection)_t = \eta P(\neg Detection|Alien_x)_t \cdot P(Alien_x)_t$$

We call this process restriction.
To summarize, we first diffuse, then follow with restriction. This allows us to accurately localize the alien.

1.2 Implementation

In this case, we use the function 'update_belief(self, beep, alien_found)' in order to update the probabilities. Here, we define a generative function. This generative function calculates the update in belief taking into account the distance from the current cell 'ci' and the bot's position 'self.pos'.

```
def update_belief(self, beep, alien_found):
        # Crew Belief
        generative_fn = lambda x: np.exp(-self.alpha*(x - 1)) if
    beep else (1 - np.exp(-self.alpha*(x-1)))
        open_cells = self.grid._grid.get_open_indices()
```

```
        for ci in open_cells:
            if ci == self.pos:
                continue
            gen_res = generative_fn(self.grid.distance(ci, self.pos
    ))
            if gen_res == 0:
                pass
            self.grid.grid[ci[1]][ci[0]].crew_belief *= gen_res
        # Normalize
        flat_beliefs = [self.grid.grid[ci[1]][ci[0]].crew_belief
    for ci in open_cells]
        belief_sum = sum(flat_beliefs)
        for ci in open_cells:
            self.grid.grid[ci[1]][ci[0]].crew_belief /= belief_sum

        self.diffuse_alien_prob(alien_found)
        self.restrict_alien_prob(alien_found)
        print("Alien detected" if alien_found else "Alien Not
    Detected")
```

If 'beep' is True, which indicates that the bot essentially senses a crew member nearby, the generative function essentially applies an exponential decay model to increase the belief in cells that are relatively closer to the bot and decrease the belief in cells farther away. This update in belief is proportional to the distance between the cells.

If 'beep' is False, the generative function adjusts this belief inversely, decreasing the crew belief in closer cells and increasing it in cells farther away.

This approach reflects the bot's reasoning that closer cells are more likely to contain the crew member if a beep is detected, while distant cells become more plausible locations if no beep is detected.

After the crew belief is updated for all open cells in the grid (excluding the bot's current position), this method proceeds to Normalize the beliefs. Here, we calculate the sum of all crew beliefs across open cells in 'belief_sum' and divide each individual belief value by this sum.
Normalization here ensures that the crew beliefs represent probabilities, with the total summing up to 1. This step maintains the relative strengths of beliefs across different cells while also ensuring they conform to a valid probability distribution.

Within the **'update_belief'** function, the **'diffuse_alien_prob'** function is called, whose main responsibility is to diffuse the probability of the presence of the alien, taking into account whether the alien has been detected or not.

```
def diffuse_alien_prob(self, alien_found):
        choose_fun = None
        if alien_found:
            choose_fun = lambda x: self.within_alien_sensor(x)
        else:
            choose_fun = lambda x: not self.within_alien_sensor(x)
        open_cells = self.grid._grid.get_open_indices()
```

```
        # Cells inside the alien sensor and just outside
        # The probability will diffuse among these
        filtered_open_cells = [oc for oc in open_cells if (
    choose_fun(oc) or self.alien_sensor_edge(oc, 1 if alien_found
    else 0) )]
        alien_belief = np.zeros((self.grid.D, self.grid.D))

        # Diffuse through the edge cells
        for ci in filtered_open_cells:
            neighbors = self.grid._grid.get_neighbors(ci)
            neighbors = [n for n in neighbors if self.grid.grid[n
    [1]][n[0]].open and choose_fun(n) ]
            # Diffuse the probability at the current square into
    the
            # neighbors that the alien can move to
            for n in neighbors:
                alien_belief[n[1]][n[0]] += self.grid.grid[ci[1]][
    ci[0]].alien_belief/len(neighbors)
        # Normalizs
        total_belief = np.sum(alien_belief)
        for ci in open_cells:
            alien_belief[ci[1]][ci[0]] /= total_belief
        # Update the original probabilities
        for ci in open_cells:
            self.grid.grid[ci[1]][ci[0]].alien_belief =
    alien_belief[ci[1]][ci[0]]
```

Here, we prepare the grid cells for diffusion by first identifying cells within the alien sensor's range or at the edge of its detection range. We then diffuse the probability among neighbouring cells, considering only the cells that are within or at the edge of the alien detection square, and divide the alien belief of the current cell (probability that an alien exists in a given cell) equally to all it's valid neighbors. We then normalize the probability distribution across all open cells to ensure that the total probability sums up to 1.

The second function called **'restrict_alien_prob'** is invoked right after the diffuse function. Here, again, we first determine whether each cell is within or outside the alien sensor's range. We then essentially identify the cells whose probabilities are to be set to zero, i.e, the cells outside the sensor range if the alien is detected or inside if not detected. Once identified, we set their alien belief to 0.0, and proceed to normalize the remaining probabilities across all open cells to ensure that the total probability sums up to 1.

```
def restrict_alien_prob(self, alien_found):
        choose_fun = None
        if alien_found:
            choose_fun = lambda x: self.within_alien_sensor(x)
        else:
            choose_fun = lambda x: not self.within_alien_sensor(x)

        open_cells = self.grid._grid.get_open_indices()
        filtered_open_cells = [oc for oc in open_cells if not
    choose_fun(oc)]
        print(f"Cells to set to 0: {len(filtered_open_cells)}")
```

```
        for ci in filtered_open_cells:
            self.grid.grid[ci[1]][ci[0]].alien_belief = 0.0
        # Normalize
        total_belief = 0
        for ci in open_cells:
            total_belief += self.grid.grid[ci[1]][ci[0]].
    alien_belief
        for ci in open_cells:
            self.grid.grid[ci[1]][ci[0]].alien_belief /=
    total_belief
```

We also define functions to implement the crew and alien sensors, conforming to the requirements set within the project.

```
     def crew_sensor(self):
        c = rd.random()
        return c <= np.exp(-self.alpha
                           * (self.grid.distance_to_crew(self.pos)
    - 1))
     def alien_sensor(self):
        found_alien = 0
        for j in range(-self.k, self.k + 1):
            if found_alien == 1:
                break
            for i in range(-self.k, self.k + 1):
                pos = [ self.pos[0] + i, self.pos[1] + j ]
                if pos[0] > self.grid.D - 1:
                    pos[0] = self.grid.D - 1
                elif pos[0] < 0:
                    pos[0] = 0
                if pos[1] > self.grid.D - 1:
                    pos[1] = self.grid.D - 1
                elif pos[1] < 0:
                    pos[1] = 0
                if self.grid.grid[pos[1]][pos[0]].alien_id != -1:
                    found_alien = 1
                    break
        return found_alien == 1
```

The **move()** method constitutes a **pivotal aspect of the bot's decision-making process**, governing its movements across the grid while factoring in both crew and alien beliefs. Here's a breakdown of the method.

At the start of every single time step, this method calls the **update_belief()** function, which updates both the crew and alien beliefs within the grid. Subsequently, it proceeds to identify the open neighboring cells of the bot, arranging them based on crew belief. This allows the bot to prioritize moving to cells where it believes crew members are more likely to be present. We then use the **'plan_path()'** function to move towards the destination cell, which is the cell with the highest crew belief, employing the use of Breadth First Search. If a path is found to this cell, the bot moves to the first cell of this path. Conversely, in the absence of a discernible path, the strategy transitions to evasion mode, guiding the bot towards the cell that exhibits the lowest alien belief instead. This adaptive approach mitigates potential encounters with the aliens. This is also the function that takes care of updating

beliefs associated with the traversed path, setting the crew and alien beliefs of the explored cells to 0, of course, assuming no crew member was found and no alien was detected.

```
    def move(self):
        self.update_belief(self.crew_sensor(), self.alien_sensor())

        neighbors = self.grid._grid.get_open_neighbors(self.pos)
        neighbors = [n for n in neighbors if not self.grid.crew_pos == n]
        neighbors.sort(key=lambda x: self.grid.grid[x[1]][x[0]].crew_belief)
        open_cells = self.grid._grid.get_unoccupied_open_indices()

        self.grid._grid.remove_bot(self.pos)
        dest_cell = max(open_cells, key=lambda x: self.grid.grid[x[1]][x[0]].crew_belief)
        self.plan_path(dest_cell)
        if len(self.path) != 0:
            self.pos = self.path[0]
        # If no path is found, we automatically shift to evasion, and the evasion strategy is basic
        # Go to the cell with the lowest alien probability
        else:
            if self.debug:
                print("Evasion!!")
            neighbors = self.grid._grid.get_neighbors(self.pos)
            open_neighbors = [n for n in neighbors if self.grid.grid[n[1]][n[0]].open]
            open_neighbors.sort(key=lambda x: self.grid.grid[x[1]][x[0]].alien_belief)
            self.pos = open_neighbors[0]
        #elif self.grid.grid[neighbors[0][1]][neighbors[0][0]].crew_belief == self.grid.grid[neighbors[-1][1]][neighbors[-1][0]].crew_belief:
        #    self.pos = rd.choice(neighbors)
        #else:
        #    self.pos = neighbors[-1]

        self.grid._grid.place_bot(self.pos)

        if self.pos != self.grid.crew_pos:
            self.grid.grid[self.pos[1]][self.pos[0]].crew_belief = 0.0

        if not self.grid._grid.has_alien(self.pos):
            self.grid.grid[self.pos[1]][self.pos[0]].alien_belief = 0.0
        self.tick += 1
```

Here is an example of the visualization, for a random run of the bot.

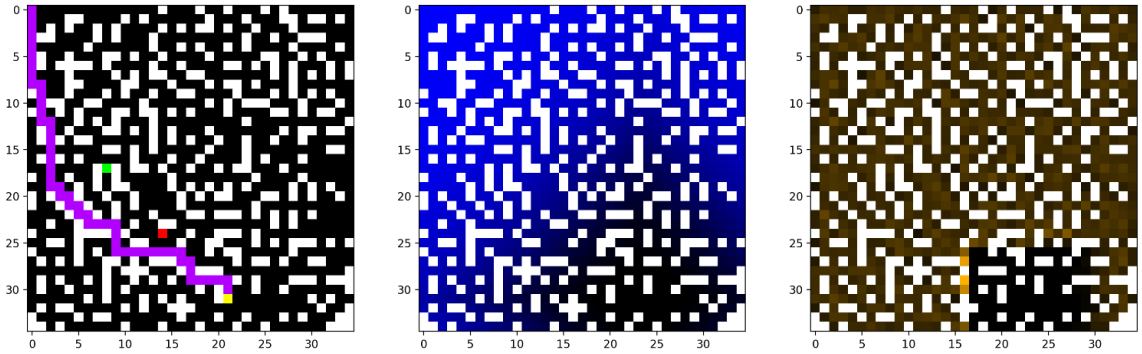

Figure 1.1: Visualization of a random run; step 3

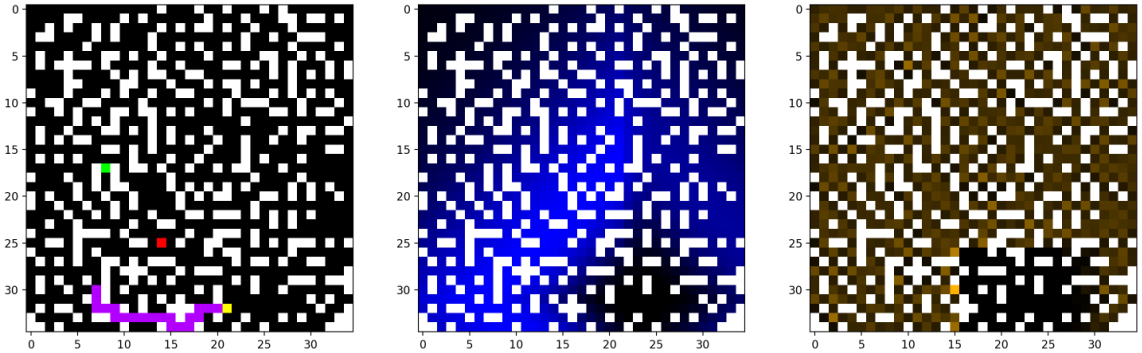

Figure 1.2: Visualization of a random run; step 4

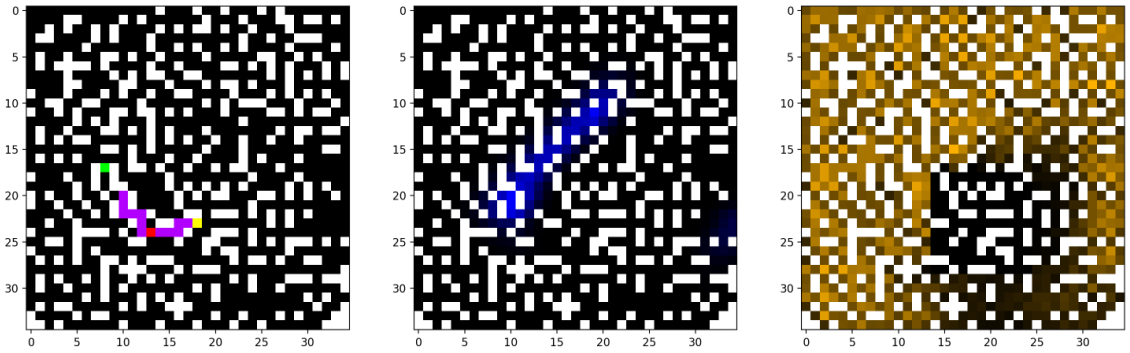

Figure 1.3: Visualization of a random run; step 100

We have three plots; the first, depicting the locations of the Crew Member (in Green), the Bot (in Yellow), and the Alien (in Red). The path (in Violet) depicts the shortest path to the cell with the highest belief in the grid.
The second grid depicts the probability distribution of the crew beliefs of this particular case (in Blue). Of course, the darker the intensity, the more the belief.
The third grid depicts the alien beliefs (in Orange), with the (2k + 1)*(2k + 1) square centered at the location of the bot.

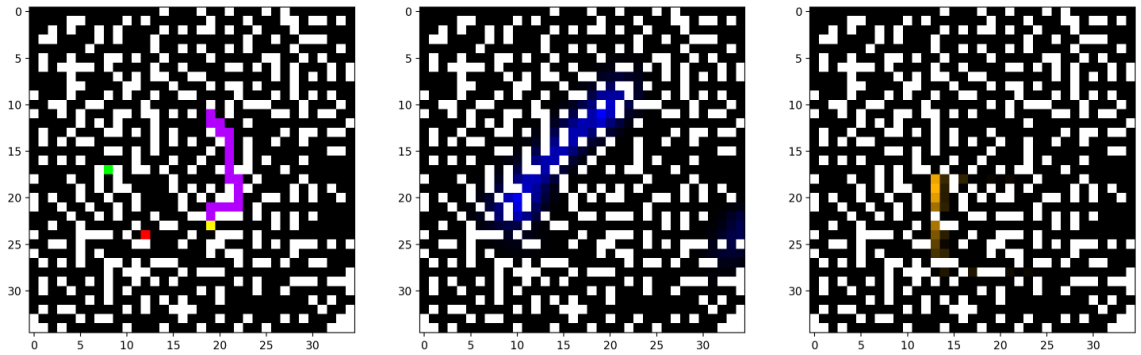

Figure 1.4: Visualization of a random run; step 101

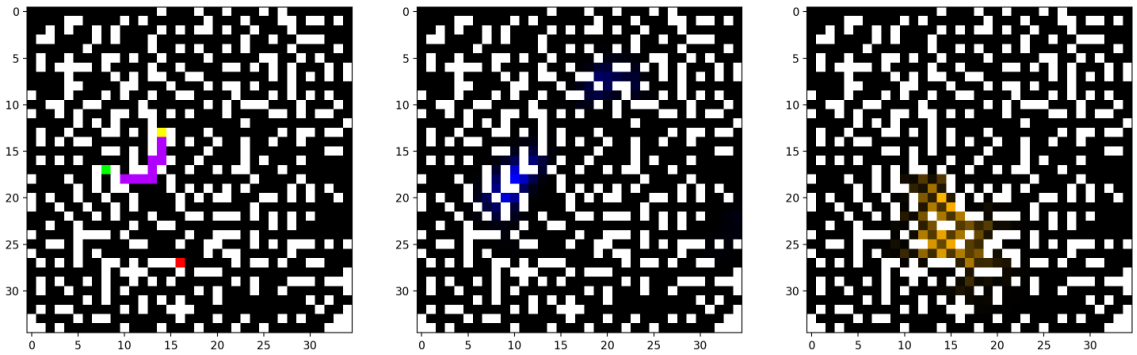

Figure 1.5: Visualization of a random run; step 160

In this particular run, the Bot receives it's first beep at step 4. After the probabilities are updated, the change in the Crew Beliefs (Second plot) can be observed from Figure 1.1 and Figure 1.2.
The Alien is within the Bot's alien detection square in Figure 1.3. The alien beliefs are consequently updated, and the change (Third plot) can be seen in Figure 1.4.
In Figure 1.5, we observe the diffusion of probabilities of the alien belief in action.

1.3 Bot 2

1.3.1 Improvements on Bot 1

The idea behind Bot 2, (and subsequently Bots 5 and 8 to a certain extent) mainly deals with a complete change in the decision-making process. Bot 1 models the world. Now that we have modeled it, how do we move so that we may gather as much information as possible in the shortest number of turns? That question paved the way for the methods we use with Bot 2. Bot 2 implements a strategy that dynamically adjusts between exploration and convergence based on the distribution of belief probabilities. It aims to explore the grid extensively before switching to a strategy focused on closing in when belief probabilities concentrate. The grid can be thought of on two levels- there exists our normal grid from before (we call this the fine grid) and a new grid of a much smaller dimension (5x5 in our case). This new grid is our coarse grid. The beep can be thought of as a 1D range sensor. If we combine measurements from multiple positions we can triangulate the position of the crew. Of course, we are working with probabilities, but the idea is still sound. The coarse grid allows us to keep track of roughly which parts of the grid have been explored and which parts have been left untouched. We then choose a random cell from the fine grid that falls within the coarse grid's cell. Exploring the grid in such a two-level manner allows us to converge to the crew member faster. But how do we know when to stop exploring? We need some method of measuring how the beliefs are distributed spatially. We chose to use a bounding box. The size of the bounding box can then be used as a measure of how concentrated the beliefs are. Once we achieve a certain concentration, we choose to go towards the cell with the highest crew belief. This method has worked out incredibly well in practice.

1.3.2 Implementation

The changes within Bot 2's initialization are as follows:

```
class bot2:
    def __init__(self, grid, alpha = ALPHA, k=5, debug=1, p=None):
        self.grid = grid
        self.pos = p
        while self.pos == self.grid.crew_pos or self.pos is None:
            self.pos = rd.choice(self.grid._grid.get_open_indices()
    )
        self.alpha = alpha
        self.debug=debug
        self.tick=0
        self.k=k
        self.DECISION_EXPLORE = 0
        self.DECISION_CLOSE_IN = 1
        self.coarse_grid_size = 7
        self.coarse_grid = [[0 for _ in range(self.coarse_grid_size
    )] for __ in range(self.coarse_grid_size)]
        self.decision_state = self.DECISION_CLOSE_IN
```

```
        self.dest_cell = None
        self.visited_cg = set()
```

To this end, we have two functions we use in this Bot.

The first function we introduce is 'measure_belief_bb_size()' which is used to calculate the size of the bounding box encapsulating the areas with significant crew belief probabilities. The maximum probability is scaled by a threshold to binarize the beliefs. The bounding box encompasses all of them. This helps in determining if the belief probabilities are concentrated enough to switch to a closing-in strategy, and this measurement also helps in understanding the spatial distribution of belief probabilities on the grid.

```
        def measure_belief_bb_size(self):
            crew_prob = [[self.grid.grid[j][i].crew_belief if self.grid.grid[j][i].open else 0 for i in range(self.grid.D)] for j in range(self.grid.D)]
            thresh = max([self.grid.grid[j][i].crew_belief for j in range(self.grid.D) for i in range(self.grid.D) if self.grid.grid[j][i].open]) * 0.2
            crew_prob_bin = [[1 if crew_prob[j][i] >= thresh else 0 for i in range(self.grid.D)] for j in range(self.grid.D)]

            # Calculate bounding box
            min_pos = [self.grid.D, self.grid.D]
            max_pos = [0, 0]
            for j in range(self.grid.D):
                for i in range(self.grid.D):
                    if self.grid.grid[j][i].open and crew_prob_bin[j][i] == 1:
                        if i >= max_pos[0] and j >= max_pos[1]:
                            max_pos[0] = i
                            max_pos[1] = j
                        if i <= min_pos[0] and j <= min_pos[1]:
                            min_pos[0] = i
                            min_pos[1] = j
            #print(f"Max pos: {max_pos}")
            #print(f"Min pos: {min_pos}")
            #print(f"Size: {self.grid.distance(min_pos, max_pos)}")
            return self.grid.distance(min_pos, max_pos)
```

The second function 'make_decision()' is used to decide how to explore. It subdivides the grid into a coarse grid (which in our case, we decided to go with 5 x 5 coarser grids) and selects random points within each cell for exploration. Again, if belief probabilities are concentrated, it switches to a closing-in strategy.

```
def make_decision(self):
        if self.decision_state == self.DECISION_EXPLORE or self.decision_state is None:
            self.decision_state = self.DECISION_EXPLORE
            #print("Exploration")
            bb_size = self.measure_belief_bb_size()
            if bb_size <= BB_SIZE:
                return self.DECISION_CLOSE_IN, None

```

```
            stride = self.grid.D / self.coarse_grid_size
            stride = int(stride)
            #print(f"Stride: {stride}")
            coarse_positions = [(i, j) for i in range(self.coarse_grid_size) for j in range(self.coarse_grid_size) if (i, j) not in self.visited_cg]
            if len(coarse_positions) == 0:
                return self.DECISION_CLOSE_IN, None
            coarse_pos = rd.choice(coarse_positions)
            self.visited_cg.add(coarse_pos)
            dest = rd.choice([(coarse_pos[0] + i, coarse_pos[1] + j) for i in range(stride) for j in range(stride) if self.grid.grid[coarse_pos[1] + j][coarse_pos[0] + i].open])
            return self.DECISION_EXPLORE, dest
        return self.DECISION_CLOSE_IN, None
```

Other helper functions used within the make_decision() function include get_coarse_pos(), which calculates the position in the coarser grid given a particular position in the original grid, and update_coarse_grid() which updates the coarse grid representation based on the current crew belief distribution in the fine grid, looping over the open cells and aggregating the crew beliefs into the corresponding cells in the coarser grid, and calculating the total belief.

```
    def get_coarse_pos(self, pos):
        stride = self.grid.D//self.coarse_grid_size
        return (pos[0] // stride, pos[1] // stride)

    def update_coarse_grid(self):
        open_cells = self.grid._grid.get_open_indices()
        tot_belief = 0
        for oc in open_cells:
            cpos = self.get_coarse_pos(oc)
            self.coarse_grid[cpos[1]][cpos[0]] += self.grid.grid[oc[1]][oc[0]].crew_belief
            tot_belief += self.grid.grid[oc[1]][oc[0]].crew_belief
        if abs( tot_belief - 1.0 ) > 1e-6:
            print(f"Total Belief is not 1!!: {tot_belief}")
```

For the time being, setting BB_SIZE to '40' yielded results (over the three performance metrics) that were far better than Bot 1, (results that we are satisfied with) but with a little more fine-tuning, there's an incredibly high chance that it gets **much** better.

1.4 Plots

Certain trends can be observed across multiple Ks. As α increases from 0, we see a sharp drop in the average number of turns needed to reach the crew members. This makes sense since α essentially sets the fall off rate of the beep. If too low, we will always get a beep, even at the opposite end of the grid. Once α has a more optimal value, we can get a much better idea of where the crew member is since the probability of getting a beep is more concentrated around the crew member. However, if α increases too much, it gets too concentrated and we spend a large

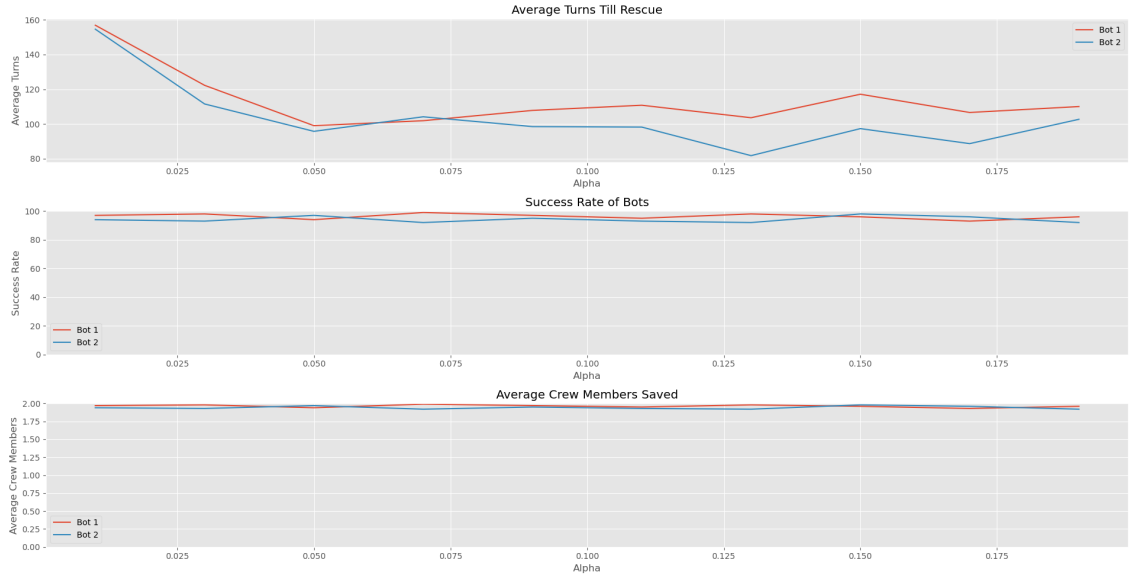

Figure 1.6: K = 5

amount of time waiting for a beep. Once we do get one, we converge fairly rapidly though.

As for the effects of K, it does not seem to affect the average number of turns to reach the crew member. It however does increase the survival rate quite a bit across the board. The larger the value of K, the lower the probability of finding an alien. More importantly, we can see that our exploration followed by closing-in strategy has paid off. Bot 2 is consistently significantly better than Bot 1. BB_SIZE can be tweaked further to improve it even more. The size used here is 40.

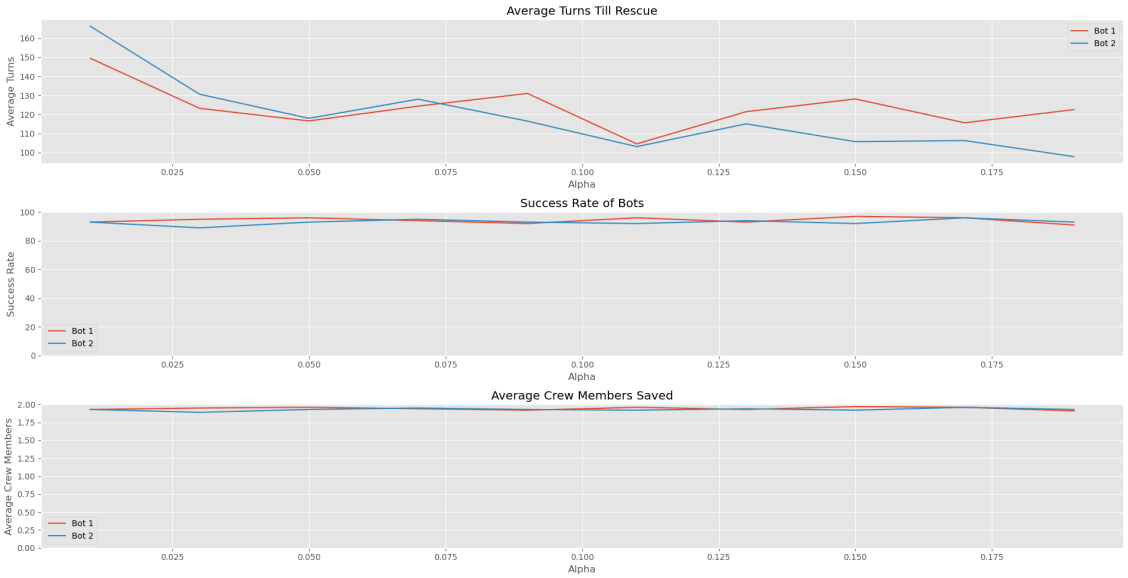

Figure 1.7: K = 10

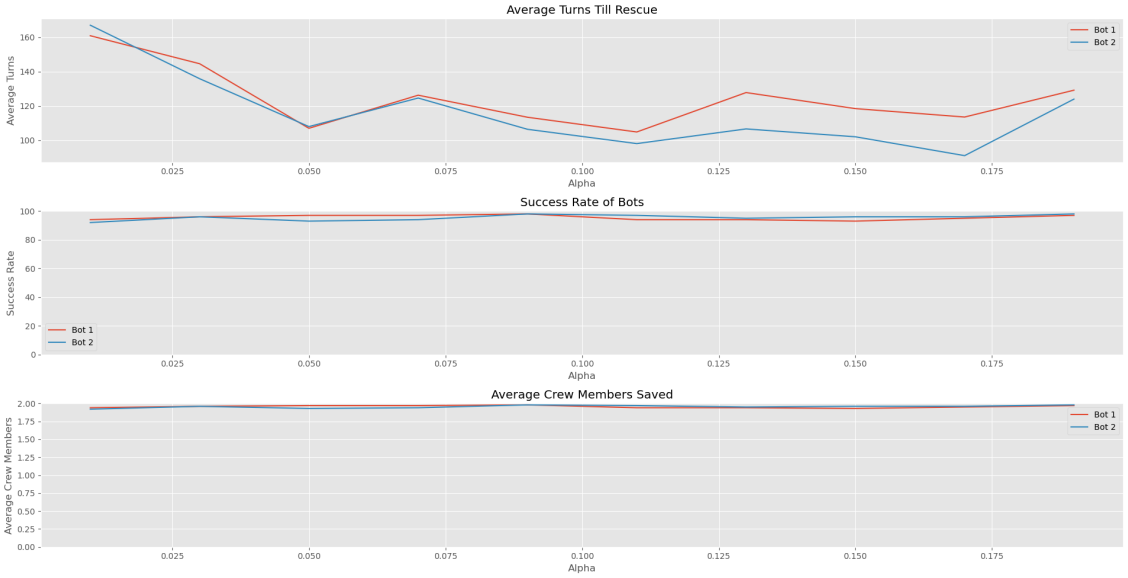

Figure 1.8: K = 15

1.5 Optimizations

1.5.1 Avoiding Needless Grid Generation

The grid is surprisingly demanding to generate. The iterative steps can go on for quite some time. If we are collecting 10 data points for a plot, and each data point involves about 100 iterations, across multiple bots, then we are generating the grid 1000s of times. This adds a fair bit of time which can be easily shaved off. Instead of generating a new grid for all the bots, we keep the same grid for the bots. This simple technique reduces the impact of grid generation by a significant amount. It can be seen in code here -

```
                ...
                self.g.reset_grid()
                self.g.crew_pos = crew_pos
                b = bot1(self.g, alpha=alpha, debug=False, p=
    bot_pos, k=self.k)
                a = Alien(self.g._grid, b, p=alien_pos)
                ...
```

As shown above, the grid remains the same across the bots, but the grid itself is reset without changing the structure. The world is then generated again. This results in a massive improvement in performance.

1.5.2 Using Slot Classes

Python's normal method of accessing attributes in a class has a decent amount of performance and memory overhead. Using slots for attributes (Can be seen in the definition of the GridAttrib class above) cuts down on this overhead.

1.5.3 Using Multiprocessing

We chose to parallelize the generation of data. The class WorldState is used for this purpose. This allows each process to have a separate copy of a Grid, Aliens, and other necessary information to run the simulations independently. If interested, refer to the function *dispatch_jobs* in the class World, and the class WorldState.

2

Bot 3, 4 and 5

The design and implementation of Bots 3, 4, and 5 are expanded upon below. In this situation, we have 2 crew members to rescue and one alien in play.

2.1 Bot 3

2.1.1 Mathematical Modeling

The modeling of Bot 3 is identical to Bot 1. Please refer to Bot 1. The one change is that once a crew member has been found the belief grid is reset.

2.1.2 Implementation

Similar to Bot 1, Bot 3 is aware of of two things at the beginning:

- It knows that the crew member is equally likely to be in any open cell other than it's initial cell.
- The alien is equally likely to be in any open cell outside the bots (2k + 1) x (2k + 1) detection square.

Every step the bot takes, it has access to two kinds of sensor data:

- Alien Sensor: Same as discussed
- Crew Sensor: Here, the bot can receive beeps from both crew members, but it can't tell which crew member it received a beep from. So in this case, the bot receives beeps like below:

$$P(Beep_1) = e^{-\alpha(d1-1)}$$
$$P(Beep_2) = e^{-\alpha(d2-1)}$$
$$P(Beep) = P(Beep_1 \cup Beep_2)$$

where,

$d1 \rightarrow$ distance from the bot to the first crew member,
$d2 \rightarrow$ distance from the bot to the second crew member,
$\alpha \rightarrow$ a positive, non-zero value

The working of this bot is very similar to that of bot 1. The only difference is that there are now two crew members and the bot could receive a beep from either one of them. The bot can't tell which crew member the beep came from. As soon as one crew member is found; they are teleported away, the probability of the grid is reset, and the bot goes on to find the second crew member.

Why do we reset the probability of the grid? How do we do it?
We reset the probability of the grid after finding the first crew member because of the nature of how our bot works. The closer the bot gets to a crew member, the more the frequency of hearing a beep increases and therefore more probability concentrates around the location of crew member 1. Once we find crew member 1, the probability of finding ***a*** crew member is still concentrated around that location while the probability everywhere else would be negligible. Since those cells are closer to the bot, it will explore all of those cells first instead of exploring the grid and trying to find crew member 2. If the bot gets lucky, then both crew members will be close to each other allowing it to discover the second crew member easily after the first, but this is seldom the case.
As for how we reset the grid, we distribute the probability of finding a crew member equally to every open cell except the ones we have already traversed. To do this, we get every cell which has its *crew_belief* not set to zero (as the *crew_belief* will only be zero once we've been to that cell and not found a crew member) and distribute the property evenly across all of those cells.

```
# This bit of code runs when we the 'found' variable is true,
# i.e. when we find our first crew member
if self.found:
    open_cells = self.grid._grid.get_open_indices()
    open_cells = [cells for cells in open_cells if self.grid.grid[
    cells[1]][cells[0]].crew_belief != 0]
    count = 0
    for cell in open_cells:
        x, y = cell
        self.grid.grid[y][x].crew_belief = 1
        count += 1

    for cell in open_cells:
        x, y = cell
        self.grid.grid[y][x].crew_belief /= count

    # set this to False so that this block of code doesn't run
    every time
    self.found = False
```

The math for updating the probability is identical to that of Bot 1 (as Bot 3 is basically an extension of Bot 1). The only difference is that we take two crew members into account when checking for the beep in our *crew_sensor()* as shown below:

```
...
class Bot3:
    def crew_sensor(self):
```

```
        c1 = rd.random()
        c2 = rd.random()
        d1, d2 = self.grid.distance_to_crew(self.pos)
        a, b = False, False

        if d1 is not None:
            a = c1 <= np.exp(-self.alpha* (d1 - 1))
        if d2 is not None:
            b = c2 <= np.exp(-self.alpha* (d2 - 1))

        return a or b
...
```

2.2 Bot 4

Again, Bot 4 is very similar to Bot 1, except that the probabilities of where the crew members are, account for the fact that there are two of them, and are updated accordingly.

2.2.1 Mathematical Modeling

We need a model of 2 crew members existing somewhere in the grid. We can do this by establishing a joint probability distribution- $P(x_1, x_2)$. We will call this $P(Crew_{x_1,x_2})$ We do not know what the probability is so we first assume that all are equally likely during initialization. We now need some way of incorporating measurements into this prior to turn it into the posterior. We rely on Bayes' Theorem-

$$P(Crew_{x_1,x_2})_{t+1} = P(Crew_{x_1,x_2}|Beep)_t = \eta P(Beep|Crew_{x_1,x_2})_t \cdot P(Crew_{x_1,x_2})_t$$

From the section on Bot 3, we know that-

$$P(Beep|Crew_{x_1,x_2}) = P(Beep_1|Crew_{x_1,x_2} \cup Beep_2|Crew_{x_1,x_2})$$

$$P(Beep|Crew_{x_1,x_2}) = e^{-\alpha(d_1-1)} + e^{-\alpha(d_2-1)} - e^{-\alpha(d_1+d_2-2)}$$

$$P(\neg Beep|Crew_{x_1,x_2}) = 1 - P(Beep|Crew_{x_1,x_2})$$

We now have the likelihoods necessary to implement our Bayesian filter. It is now just a matter of letting our probability distribution tell us where we can find both crew members. Of course, since the state space for this model is much larger, it is computationally more expensive than before.

2.2.2 Implementation

The main difference between Bot 3 and Bot 4 is accounting for the fact that there are two crew members. It makes no difference to Bot 3 how many crew members are present or how many sources it receives a beep from. It receives a beep, updates the probability accordingly, and moves towards the cell with the highest belief.

Bot 4 accounts for there being two crew members by calculating the probability of each possible combination of crew member positions and heading to the closest cell of the combination with the highest probability. "What if crew member 1 is at location (0, 0) and crew member 2 is at location (0, 1)?" "What about (0, 0) and (0, 2)?" Like this, it stores a list of all permutations of possible crew member position pairs and calculates the probability for each of them. Each step it takes, it rules out certain combinations allowing it to quickly find the second crew member as soon as it finds the first.

That sounds nice, but how do you implement it for Bot 4?
We implement this using a Python Dictionary.

```
class Grid2:
    def __init__(self, ...):
        ...

        # defining the beliefs dictionary
        self.beliefs = {}
        # get open cells not occupied by the bot
        open_cells = self._grid.get_unoccupied_open_indices()

        # total number of combinations
        sum = len(open_cells) * (len(open_cells) - 1) / 2
        for one_cell in open_cells:
            for two_cell in open_cells:
                # because the order of crew members don't matter
                if (two_cell, one_cell) in self.beliefs or \
                        one_cell == two_cell:
                    continue

                # equal distribution
                self.beliefs[(one_cell, two_cell)] = 1 / sum
```

The dictionary, as described, stores beliefs for every combination of open cells (which is where the crew members might be). Let's say the bot is currently at (10, 14) and moves to (10, 15) finding no crew member at that location. It eliminates every combination in the dictionary that contains (10, 15) coordinate in the key. Since we know for a fact that there's no crew member at that location, it's probability immediately becomes 0.

```
def move(self):
    if (self.pos != self.found_crew) and \
        (self.pos != self.grid.crew_pos) and (self.pos != self.grid
    .crew_pos2):
        self.grid.grid[self.pos[1]][self.pos[0]].crew_belief = 0.0
        keys_to_delete = []
        for key, _ in self.grid.beliefs.items():
            if self.pos in key:
                self.grid.beliefs[key] = 0
                keys_to_delete.append(key)
        for k in keys_to_delete:
            del self.grid.beliefs[k]
```

That's how we store beliefs, and this is what our bot updates and uses at every step to make sure it's headed in the direction of highest probability.
We've now figured out how to store beliefs for the way that Bot 4 works. With this structure and our previously established knowledge of probability, we're now going to update the beliefs in this dictionary to steer the bot in the right direction. We do it as simply as possible. We go through every single key in our dictionary (which corresponds to every combination of 2 cells with non-zero belief of crew members existing there) and apply the equations from the section on modeling. We keep on iterating and we slowly converge on the actual probability. This is carried out in the function - *update_belief*. The relevant part of the function is shown below.

```
for key, _ in self.grid.beliefs.items():
    generative_fn = lambda x: exp(-self.alpha * (x - 1)) \
        if beep else (1 - (exp(-self.alpha * (x - 1))))

    one_cell, two_cell = key
    gen_crew_one, gen_crew_two = 0, 0

    gen_crew_one = generative_fn(self.grid.distance(one_cell, self.pos))
    gen_crew_two = generative_fn(self.grid.distance(two_cell, self.pos))
    if beep:
        total_prob = gen_crew_one + gen_crew_two - gen_crew_one * gen_crew_two
    else:
        total_prob = gen_crew_one + gen_crew_two - gen_crew_one * gen_crew_two
        total_prob = 1 - total_prob
    self.grid.beliefs[(one_cell, two_cell)] *= total_prob


# Normalize
sum_beliefs = sum(self.grid.beliefs.values())
for key, value in self.grid.beliefs.items():
    self.grid.beliefs[key] = value / sum_beliefs
```

2.3 Bot 5

2.3.1 Improvements on Bot 4

Like Bot 2, Bot 5 does not overhaul the math, but rather improves on how we move around the grid so that the beliefs converge more quickly. The strategy is very similar to Bot 2. We divide the grid into a coarser grid and explore the grid till a measured bounding box is small enough in size. Once we reach this size threshold, we switch to a "closing-in" strategy. We plan the path to the nearest of the highest belief pair. For a more detailed description of how this happens, please refer to Bot 2.

2.3.2 Implementation

Since the traversal of the grid and the update of the beliefs are covered in Bot 2 and Bot 4 respectively, we will not cover them here. We will focus on the changes. The most important change happens to the function *measure_belief_bb_size*. This function measures the bounding box size, as its name suggests. The issue, of course, is that we are now in a 4D space instead of a 2D space. We need to think how we can get something that approximates the size of a bounding box. More precisely, this needs to be a quantity that increases in proportion to how spread out the beliefs are. We first start out with a function that can give us a sense of distance in this 4D space-

```
...
    def distance_4d(self, p1, p2):
        p1_1, p1_2 = p1
        p2_1, p2_2 = p2
        d1 = self.grid.distance(p1_1, p2_1) + self.grid.distance(
    p1_2, p2_2)
        d2 = self.grid.distance(p1_1, p2_2) + self.grid.distance(
    p1_2, p2_1)
        return min(d1, d2)
...
```

This distance is calculated in a simple manner. We add the Manhattan distances between the two position-pairs. We then calculate it with one of the pairs flipped. We return the smaller of the two. This is a sufficient distance function, as we will see. Now that we have a distance function, we can understand how our main measurement function works. The code is-

```
...
    def measure_belief_bb_size(self):
        highest_prob_pair = max(self.grid.beliefs.keys(), key=
    lambda x: self.grid.beliefs[x])
        highest_prob = self.grid.beliefs[highest_prob_pair]
        thresh = 0.3
        binarized_pairs = [k for k in self.grid.beliefs.keys() if
    self.grid.beliefs[k] > highest_prob*thresh]
        max_dist = 0
        for pair in binarized_pairs:
            dist = self.distance_4d(highest_prob_pair, pair)
            if dist > max_dist:
                max_dist = dist
        return max_dist
...
```

Much like before, we first binarize the entire 4D space by using a threshold multiplied by the maximum probability across the entire input space. Now that we have something easier to handle, we calculate the maximum distance from the pair with highest probability to all other pairs that survived the previous thresholding. This maximum distance is our measure of how spread out the probabilities are, in 4D space. We can now compare this with another threshold to decide the behavior. This can be seen in the function *move*. Relevant bits of the code are below-

```
...
    def move(self):
        self.update_belief(self.crew_sensor(), self.alien_sensor())
        bb_size = self.measure_belief_bb_size()
        print(f"Bounding Box size: {bb_size}")
        if bb_size <= BB_SIZE_4D:
            if self.decision_state == self.DECISION_EXPLORE:
                print(f"Turns till convergence: {self.tick}")
            self.decision_state = self.DECISION_CLOSE_IN
        ...
...
```

Similar changes are also made to the *make_decision* function, but the main job of that function is to decide which cell to explore next, which remains identical to Bot 2.

2.4 Plots

Much of what was discussed for the first two bots still applies here. There is generally a drop as α increases. The average number of turns then start increasing again once α is so high that the discoverability of the crew members drop. Another thing to note is the significantly higher number of turns required across the board compared to the previous bots. We also find that Bot 4 and Bot 5 behave better than Bot 3 consistently across varied conditions. Bot 5 is generally better than Bot 4, however for larger values of K, we could tune BB_SIZE_4D better.

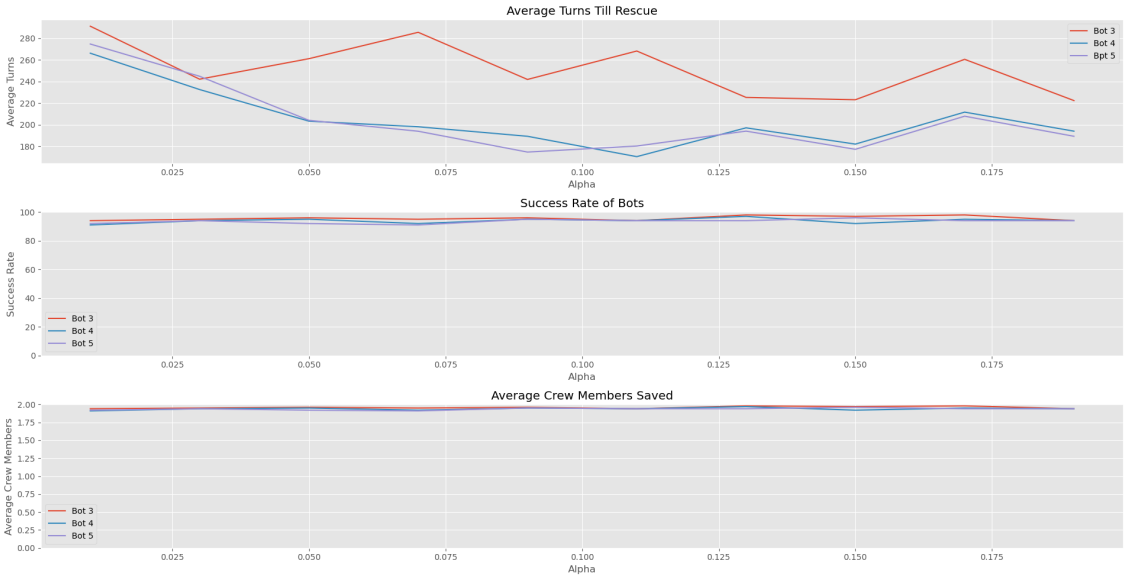

Figure 2.1: K = 5

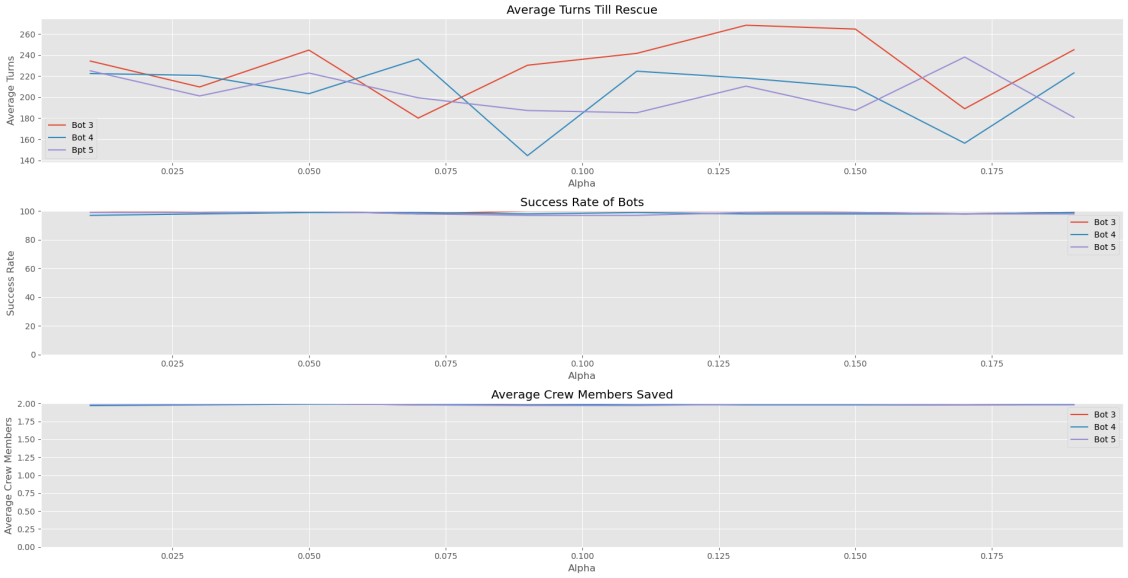

Figure 2.2: K = 10

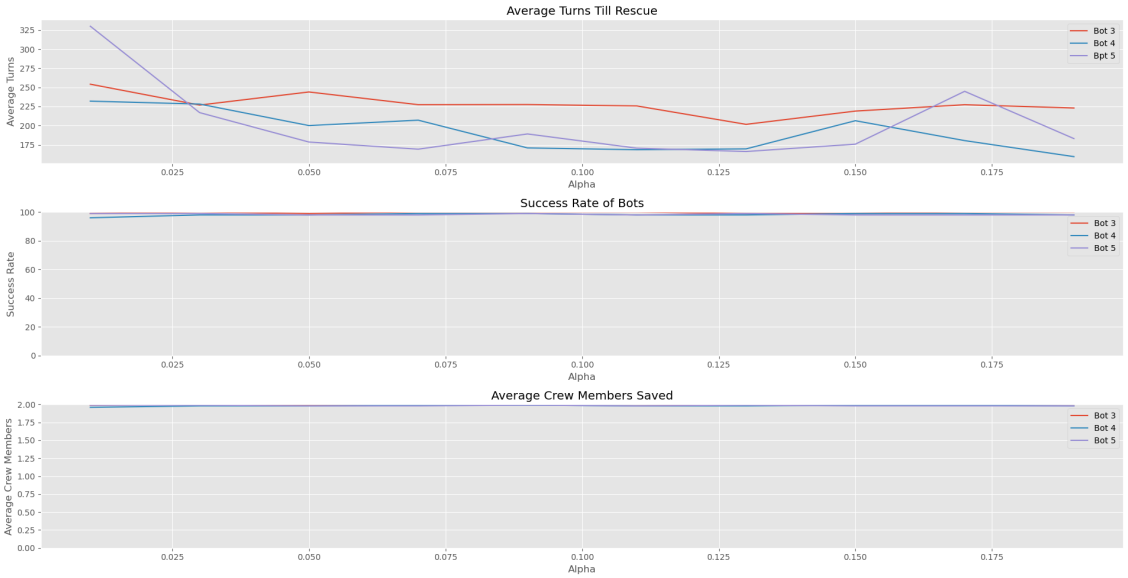

Figure 2.3: K = 15

3
Bot 6, 7 and 8

The design and implementation of Bots 6, 7, and 8 are expanded upon below. In this situation, we have 2 crew members to rescue and two aliens in play.

3.1 Bot 6

3.1.1 Mathematical Modeling

The modeling of Bot 6 is identical to Bot 3, where there's two crew members to rescue, and as the first crew member is found, they are teleported away and the updates continue until the second crew member is also found. The difference lies in the fact that there are now two aliens instead of one, and that positively identifying a cell as not containing an alien does not mean that the other squares do not have aliens.

3.1.2 Implementation

Again, the implementation of the crew mate probabilities remains the same as it is for Bot 3, but the way the alien probabilities are calculated changes.

In the scenario where a single alien was present, our detection mechanism involved resetting the alien belief probabilities of all cells outside the detection square to zero upon identifying the alien within the alien detection square. However, with the introduction of a second alien, the certainty of one alien's presence within the detection square does not inherently negate the possibility of the other alien's presence elsewhere, as we lack precise information regarding its location. To handle this case, we've made a couple of adjustments.

Specifically, within the function 'restrict_alien_prob()', we have refrained from resetting the alien belief probabilities of cells outside the detection square to zero following the identification of an alien within the detection square. This adjustment ensures that our algorithm accounts for the possibility of multiple aliens.

Again, while this modification alters how alien beliefs are updated, the general implementation of alien belief model remains consistent with the same model we've been using since Bot 1.

```
1 def restrict_alien_prob(self, alien_found):
```

```
        open_cells = self.grid._grid.get_open_indices()
        choose_fun = None
        if alien_found:
            choose_fun = lambda x: True #Dummy lambda to account
    for 2 aliens
        else:
            choose_fun = lambda x: not self.within_alien_sensor(x)
            filtered_open_cells = [oc for oc in open_cells if not
    choose_fun(oc)]
            for ci in filtered_open_cells:
                self.grid.grid[ci[1]][ci[0]].alien_belief = 0.0
        # Normalize
        total_belief = 0
        for ci in open_cells:
            total_belief += self.grid.grid[ci[1]][ci[0]].
    alien_belief
        for ci in open_cells:
            self.grid.grid[ci[1]][ci[0]].alien_belief /=
    total_belief
```

3.2 Bot 7

Bot 7 is essentially Bot 4, with the added complexity of accounting for there existing two aliens along with two crew mates. Since the two bots are identical, we shall only discuss the details of modeling the aliens.

3.2.1 Mathematical Modeling

Everything concerning how the crew members' locations are estimated can be found in the chapters on Bot 4. We will concentrate on the changes in modeling the aliens. Since there are two aliens now, we use a joint probability for aliens too. We denote this by $P(Alien_{x_1,x_2})$. The idea of diffusion and restriction still applies. The question is, how does it work in this new joint distribution? The diffusion is simple enough. The idea is to distribute the probability related with the current pair into whatever pairs can be transitioned to in the future. Let us say that the neighborhoods are $N_A(x_1)$ and $N_A(x_1)$. Every possible pair that can created by choosing one element from the first neighborhood and the second from the second is now our new 2D neighborhood. We will denote this with $N_A(x_1, x_2)$. Our new diffusion step can be put the following way in terms of probability-

$$P(Alien_{x_{n1},x_{n2}})_{t+1} = \frac{P(Alien_{x_1,x_2})_t}{n(N_A(x_1,x_2)_t)},\ x_{n1}, x_{n2} \in N_A(x_1,x_2)_t$$

$$P(Alien_{x_1,x_2})_{t+1} = 0$$

Of course, the above needs to be superposed over other results so we don't throw away contributions from other pairs. The biggest change is to restriction. It is now in 4 dimensions as opposed to 2. Let us have a look-

$$P(Detection|Alien_{x1,x2}) = \begin{cases} 1 & x1 \text{ or } x2 \in \text{Detection Square} \\ 0 & x1 \text{ and } x2 \notin \text{Detection Square} \end{cases}$$

$$P(\neg Detection|Alien_{x1,x2}) = 1 - P(Detection|Alien_{x1,x2})$$

Unlike before, one alien being detected does not mean we can rule out another alien existing outside the detection square. In 4D however, we can rule out pairs using the above given likelihoods. This allows us to prune states that are not possible. This method is computationally expensive, of course, but it allows us to track aliens in a better fashion. Now, since we have our likelihood, we can use Bayes' Theorem to compute our posterior-

$$P(Alien_{x1,x2}|Detection)_t = \eta P(Detection|Alien_{x1,x2})_t \cdot P(Alien_{x1,x2})_t$$

$$P(Alien_{x1,x2}|\neg Detection)_t = \eta P(\neg Detection|Alien_{x1,x2})_t \cdot P(Alien_{x1,x2})_t$$

All that remains is to avoid cells with significant alien probability on the way to our crew member.

3.2.2 Implementation

Much of Bot 7 and Bot 4 are common. For other implementation details, please refer to Bot 4. We will only cover the relevant changes here surrounding alien modeling. The two functions as before are *diffuse_alien_belief* and *restrict_alien_belief*. Let's look at them one by one-

```
        ...
        new_alien_belief = {}
        for k in self.alien_beliefs.keys():
            new_alien_belief[k] = 0
        for k, v in self.alien_beliefs.items():
            if v == 0:
                continue
            npairs = self.get_neighboring_pairs(k)
            if alien_found:
                npairs = [npair for npair in npairs if self.
    within_alien_sensor(npair[0]) or self.within_alien_sensor(npair
    [1])\
                          or self.alien_sensor_edge(npair[0], 1) or
    self.alien_sensor_edge(npair[1], 1)]
            else:
                npairs = [npair for npair in npairs if not(self.
    within_alien_sensor(npair[0]) or self.within_alien_sensor(npair
    [1]))\
                          or self.alien_sensor_edge(npair[0], 0) or
    self.alien_sensor_edge(npair[1], 0)]
            for npair in npairs:
                if npair not in self.alien_beliefs:
                    if (npair[1], npair[0]) in self.alien_beliefs:
                        npair = (npair[1], npair[0])
                    else:
                        print("SOMETHING IS WROOONG!")
                        print(f"Offending key: {npair}")
                        exit(-1)
                new_alien_belief[npair] += v/len(npairs)
        self.alien_beliefs = new_alien_belief
...
```

Let's go through this step by step. We first create a new dictionary and initialize all possible pairs to 0. This is done to allow for superposition. We then iterate through all key-value pairs in the dictionary. If we find that the belief is 0 for that cell pair, then we skip it since there is no point distributing those probabilities. We then get the neighboring pairs. These pairs are then filtered according to whether both the aliens or outside, inside or only one is inside. If the sensor detects an alien, all pairs that have both cells outside of the detection square get pruned, the only exception being those that exist on the outer edge, just like for Bot 1 and the rest. If the sensor does not detect an alien, the ones satisfying the reverse condition get pruned. We then iterate through all of the neighbors and distribute the probability among them. Now let us move on to restriction-

```
        ...
        for k in self.alien_beliefs.keys():
            cell, cell2 = k
            if alien_found:
```

```
5                   if not(self.within_alien_sensor(cell) or self.
    within_alien_sensor(cell2)):
6                       self.alien_beliefs[k] = 0
7               else:
8                   if self.within_alien_sensor(cell) or self.
    within_alien_sensor(cell2):
9                       self.alien_beliefs[k] = 0
10  ...
```

We iterate through all the keys in the dictionary. Depending on whether the alien is found or not, we zero out corresponding beliefs. If an alien is found in our detection square, we know for sure that both aliens do not fall out of the detection square. So we zero out pairs that fall out of the detection square completely. Vice versa if an alien is not found.

3.3 Bot 8

3.3.1 Improvements on Bot 7

Similar to Bot 2 and Bot 5, Bot 8 uses the mathematical models of Bot 7 and builds on it with a more sophisticated decision-making system. Here, we use Bot 5's method of exploring the grid and then zoning in. The functions used here are identical to the ones used in Bot 5. Please refer to Bot 5 for a more detailed explanation.

3.3.2 Implementation

We will go through the relevant bits of code. This will be a simple reiteration since much of the code is derived from previous bots. We will go through the code that relates to coarse grid traversal. This is identical to Bot 5.

```
        ...
             stride = self.grid.D / self.coarse_grid_size
            stride = int(stride)
            coarse_positions = [(i, j) for i in range(self.
    coarse_grid_size) for j in range(self.coarse_grid_size) if (i, j
    ) not in self.visited_cg]
            if len(coarse_positions) == 0:
                return self.DECISION_CLOSE_IN, None
            coarse_pos = rd.choice(coarse_positions)
            self.visited_cg.add(coarse_pos)
            dest = rd.choice([(coarse_pos[0] + i, coarse_pos[1] + j
    ) for i in range(stride) for j in range(stride) if self.grid.
    grid[coarse_pos[1] + j][coarse_pos[0] + i].open])

        ...
```

The code above tells us how we choose our next destination. We randomly explore a coarse cell that has not been explored so far. We then add the coarse cell to the set *visited_cg* so that we may not go there again. If we are out of coarse cells to choose from, we close in on the highest belief regardless. If that is not the case, we then choose any random open cell within this coarse cell and travel there. This function is repeated every time the bot reaches its chosen destination till we have explored the entire grid or the bounding box is small enough. The bounding box size threshold is set by *BB_SIZE_4D*. The ideal value for this parameter changes with α since α determines how spread out the beliefs are over time. The current implementation uses a static parameter. A simple method to improve this might be a linear model that chooses *BB_SIZE_4D*.

```
        ...
             stride = self.grid.D / self.coarse_grid_size
            stride = int(stride)
            coarse_positions = [(i, j) for i in range(self.
    coarse_grid_size) for j in range(self.coarse_grid_size) if (i, j
    ) not in self.visited_cg]
            if len(coarse_positions) == 0:
                return self.DECISION_CLOSE_IN, None
```

```
            coarse_pos = rd.choice(coarse_positions)
            self.visited_cg.add(coarse_pos)
            dest = rd.choice([(coarse_pos[0] + i, coarse_pos[1] + j
) for i in range(stride) for j in range(stride) if self.grid.
grid[coarse_pos[1] + j][coarse_pos[0] + i].open])

        ...
```

We shall once more revisit the *move* function-

```
        ...
        beep = self.crew_sensor()
        alien_found = self.alien_sensor()
        self.update_belief(beep, alien_found)
        bb_size = self.measure_belief_bb_size()
        print(f"Bounding Box size: {bb_size}")
        if bb_size <= BB_SIZE_4D:
            if self.decision_state == self.DECISION_EXPLORE:
                print(f"Turns till convergence: {self.tick}")
            self.decision_state = self.DECISION_CLOSE_IN
        if self.pos == self.dest_cell and self.decision_state ==
self.DECISION_EXPLORE:
            self.decision_state, self.dest_cell = self.
make_decision()
        if self.decision_state == None:
            self.decision_state, self.dest_cell = self.
make_decision()
        ...
```

The above is used to decide whether to keep on exploring or to start closing in. We simply measure the bounding box and use that to decide whether the beliefs are concentrated enough. If so, we stop exploration and start closing in.

```
        if self.decision_state == self.DECISION_CLOSE_IN:
            neighbors = self.grid._grid.get_open_neighbors(self.pos
)
            neighbors.sort(key=lambda x: self.grid.grid[x[1]][x
[0]].crew_belief)
            open_cells = self.grid._grid.
get_unoccupied_open_indices()

            self.grid._grid.remove_bot(self.pos)

            if self.switch_to_single:
                dest_cell = max(open_cells, key=lambda x: self.grid
.grid[x[1]][x[0]].crew_belief)
                print(f"Dest Cell: {dest_cell}, Actual Crew: {self.
grid.crew_pos}, Current Pos: {self.pos}, Distance to Crew: {self
.grid.distance_to_crew(self.pos)}")
            else:
                max_belief = max(self.grid.beliefs.values())
                sorted_positions = [key for key in self.grid.
beliefs.keys()]
                sorted_positions.sort(key=lambda x: self.grid.
beliefs[x])
                position = sorted_positions[-1]
                dest_cell = min(position[0], position[1],
```

```
17                                  key=lambda x: abs(x[0] - self.pos
    [0]) + abs(x[1] - self.pos[1])
18                              ) if self.found_crew is None else (
    position[0] if self.found_crew == position[1] else position[1])
19                  sorted_alien_pos = [key for key in self.
    alien_beliefs.keys()]
20                  sorted_alien_pos.sort(key=lambda x: self.
    alien_beliefs[x])
21                  if self.debug:
22                      print(f"No. of pairs: {len(self.grid.beliefs)}"
    )
23                      print(f"Top 3 position pairs: {sorted_positions
    [-1]} : {self.grid.beliefs[sorted_positions[-1]]}, {
    sorted_positions[-2]} : {self.grid.beliefs[sorted_positions
    [-2]]}, {sorted_positions[-3]} : {self.grid.beliefs[
    sorted_positions[-3]]}")
24                      print(f"Dest Cell: {dest_cell}, actual crew: {
    self.grid.crew_pos}, {self.grid.crew_pos2}")
25                      print(f"Alien positions pairs: {
    sorted_alien_pos[-1:-4:-1]}")
26              self.plan_path(dest_cell)
27              if len(self.path) != 0:
28                  self.pos = self.path[0]
29              # elif self.grid.grid[neighbors[0][1]][neighbors
    [0][0]].crew_belief == self.grid.grid[neighbors[-1][1]][
    neighbors[-1][0]].crew_belief:
30                  # self.pos = rd.choice(neighbors)
31              else:
32                  print("Unable to plan! Evading!")
33                  neighbors = self.grid._grid.get_neighbors(self.pos)
34                  open_neighbors = [n for n in neighbors if self.grid
    .grid[n[1]][n[0]].open]
35                  open_neighbors.sort(key=lambda x: self.
    compute_1alien_belief(x))
36                  self.pos = open_neighbors[0]
37              self.grid._grid.place_bot(self.pos)
38
39          ...
```

The above bit of code is how we choose to navigate the environment when we are closing in. We first check to see if *switch_to_single* is true. This flag tells us if a crew member has already been rescued. If so, the movement is simple - go to the cell with the highest probability. If not, we must first sort all the possible pairs ordered in an ascending manner with respect to the probability of it being the configuration of crew members. We then take the most probable one and from that we choose the position closes to us. We then plan our path to this cell. Please keep in mind that *plan_path* will make sure that the first step that we take does **not** have an alien. It will only proceed if the first cell in the path has an alien probability of 0. If it is unable to find one, we start evasion. We evade by moving into the cell where the alien is least likely to be. It is a simple avoidance algorithm, but considering the very low alien captures we experience, we thought it would be best to focus on other parts of the bot rather than go with a more complicated evasion/path planning algorithm.

3.4 Plots

One thing to note is that for the last two plots, we decreased the number of values of alpha for which simulations were done. These last two bots are computationally very heavy and would take on the order of half a day if we were to plot it with the same number of data points as the first plot.
Again, much of what was discussed previously regarding α and K still applies. This time, however, the difference between our bots isn't quite as wide as before. Bot 7 and Bot 8 generally perform better than Bot 6. Bot 8 is better than Bot 7, but not by much, unlike the last two bouts. It may be ideal to switch to a dynamically chosen BB_SIZE_4D. A riskfactor based exploration method might be a good choice too.

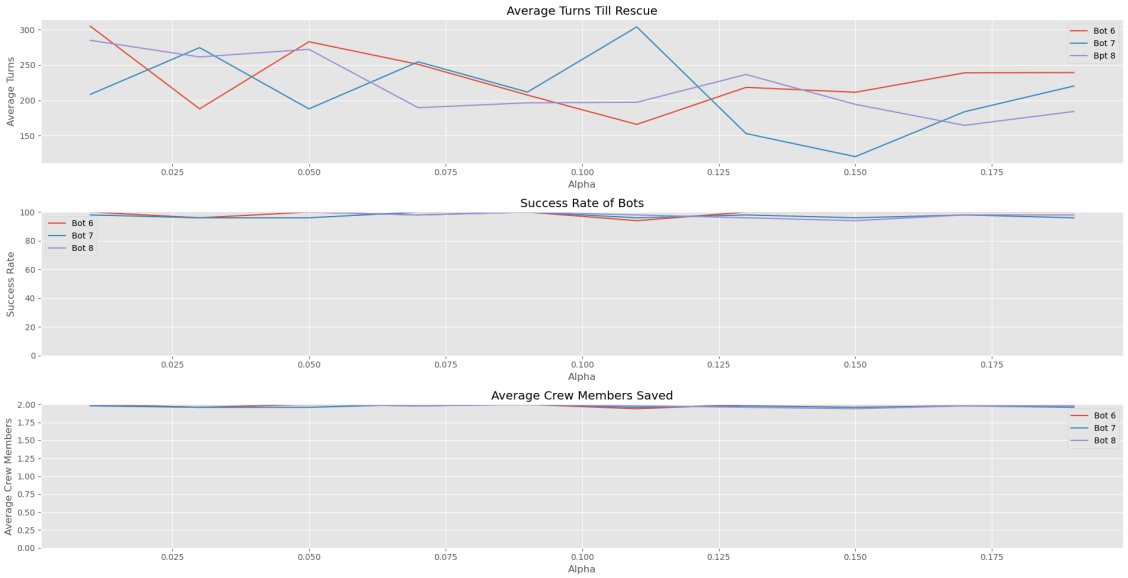

Figure 3.1: K = 5

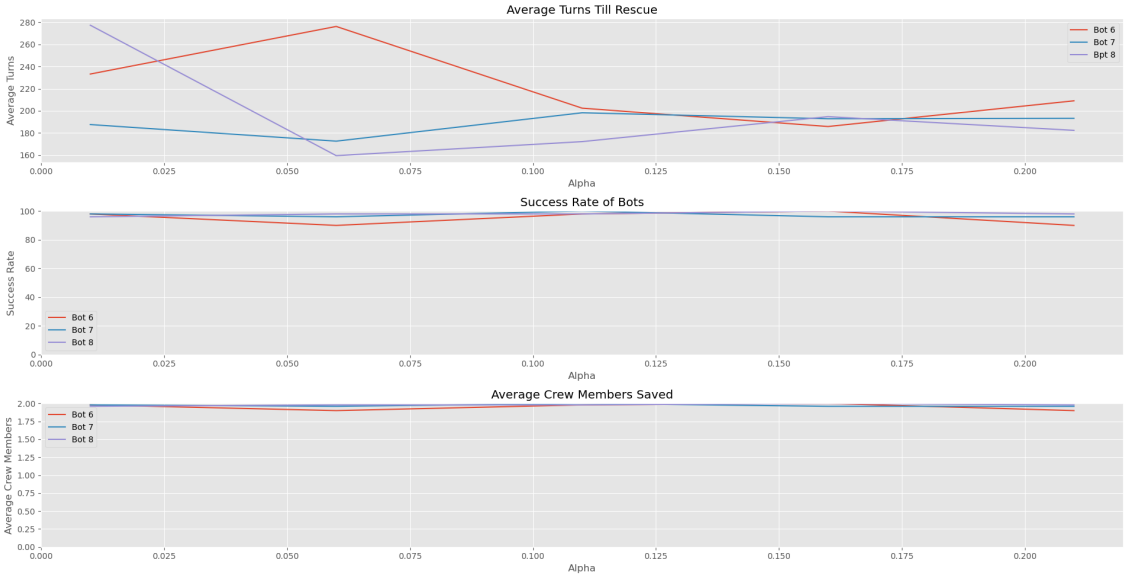

Figure 3.2: K = 10

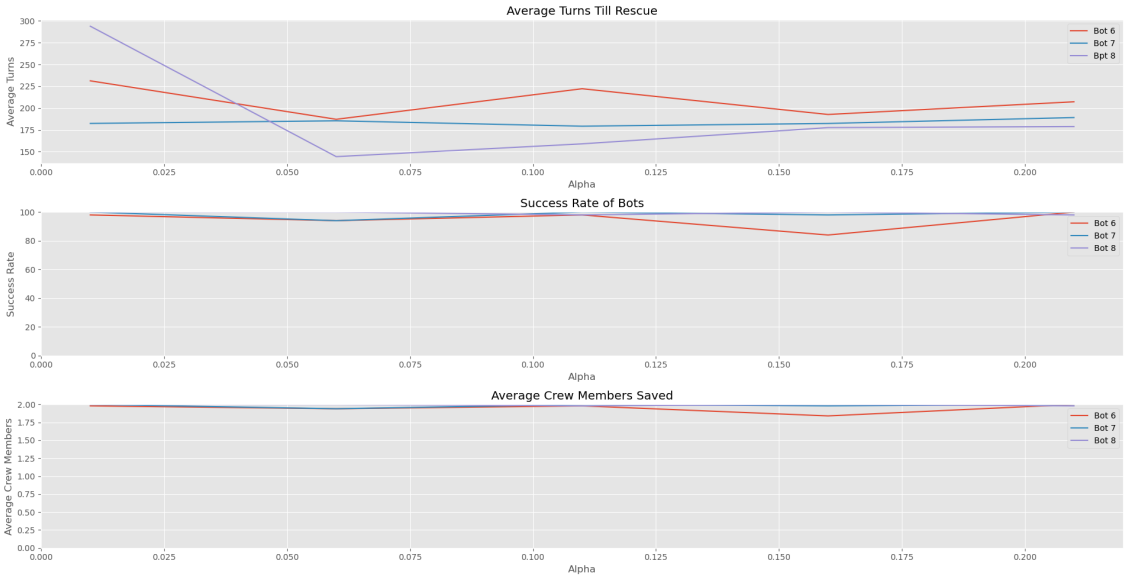

Figure 3.3: K = 15

4

Ideal Bot and Bonus Question

4.1 Ideal Bot

Designing the ideal bot for rescuing crew members and evading aliens can be done by employing a Markov Decision Process (MDP). This mathematical framework enables modeling decision-making in scenarios where outcomes are influenced by both random factors and the decisions made by an agent. The MDP process can be represented by using 4 tuples:

The State Space: In this case, essentially representing the grid, including the aliens, crew mates, and the bot.

The Action Space: In this case, it would be representing the possible actions the bot can take, i.e., moving up/down/left/right.

Transition Probabilities: This would include the probabilities associated with moving from one state to another given an action, also taking into consideration the uncertainty involved in the locations of the crew mates and the aliens.

Immediate Reward: Rewards will be allocated to various actions, in this case, say avoiding an alien successfully or finding the crew mate successfully.

The system maintains a comprehensive overview of the game environment, containing confirmed as well as estimated alien and crew locations. The Bot, accessing data on various path success rates and strategies, prioritizes actions based on their likelihood of success.

Here, we would develop a transition model by leveraging historical data to forecast the consequences of various actions. We would then compute an optimal policy to determine the best action at each state for maximum cumulative reward. We then calculate the value of each state to estimate the potential total reward achievable from that point onward. We finally determine Q-values for state-action pairs, indicating the expected utility of taking a specific action and following the optimal policy thereafter.

This bot would balance exploration of uncharted areas to locate crew members while exploiting known safe routes. It assesses the risk associated with potential actions,

incorporating a risk-aware approach to minimize encounters with aliens. Using reinforcement learning, it adapts its strategies, continuously improving performance based on past outcomes.

4.2 Bonus Question

The unpredictability of the crew members moving randomly by one cell per timestep introduces a dynamic element to the problem. This alteration would substantially affect the algorithms employed by the bots and their overall efficiency in navigating and locating crew members.
In this case, the bots would have to also account for the possible movements of the crew members. One way of doing this is to diffuse the probabilities of the crew members the same way we do for the aliens. We could maybe track the crew members better if the crew member moved in a particular manner but since the movement is random we have no choice. We can measure for a beep, calculate the beliefs for each cell, and then diffuse those beliefs for the next prior. Another way of taking care of this would be to move towards the last known position (position with the highest belief after hearing a beep) and performing a local search once it reaches the position.
Both methods would undoubtedly increase the average number of moves needed to rescue all crew members in most cases since the beliefs will take a higher number of moves before it converges.
The probability of successfully avoiding the alien and rescuing the crew would also likely decrease, simply due to the higher number of moves for which the bot stays on the grid, and the bots would require additional logic to avoid aliens taking into account the random movements of the crew.
The average number of crew members saved is also likely to decrease for the same reason.

5
Contributions

The project was equally contributed to by all three members. Every step from brainstorming ideas to implementing code was done together.