Patryk Dziedzic and Mathew Umano
Dr. Cowan
CS440 - Introduction to Artificial Intelligence
15 November 2023

Project 2 Writeup - Data and Analysis

1. Explain the design and algorithm for the bots you designed, being as specific as possible as to what your bots are actually doing. How do your bots factor in the available information (deterministic or probabilistic) to make more informed decisions about what to do next?

Deterministic Bots

Bot 1

- Initialization
    - Bot 1 initializes the bot to a random open cell. It initializes the leak to be a random open cell that is outside of the bot's detection square. Then the bot's noLeak attribute is set to true as well as all the cells in the detection square. This indicates there is no leak at those cells. Cells with noLeak = false indicate the leak *might* be at that cell.
- While the bot is not the leak:
    - The algorithm will calculate the BFS shortest path from the bot to the nearest potential leak in the Bfs.detSP_BFS() method. It will move the bot to that cell using the calculated shortest path. If the leak was found, return. If not, we perform a sense in detSenseAction(). Loop until the leak is found.
- detSenseAction()
    - This action gets the detection square and checks to see if the leak is in the square. If it's not, set all cells in the square to have noLeak = true. Otherwise, a leak has been detected. The algorithm will set noLeak = true for everything outside of the square.
- getDetectionSquare()

- The detection square is calculated to be all cells within a k radius of the bot, so the square is (2k+1)x(2k+1).

Bot 2

- Bot 2 has the same procedure as Bot 1, but it senses *at every step*. In Bot 1, we used a moveBot() method to move the bot from its current location to the nearest cell that has noLeak = false. It then performed a sense when it got there. Bot 2 does not use moveBot(). It instead only moves once in the direction of the calculated shortest path. This allowed Bot 2's smallest number of actions to be less than Bot 1's smallest number of actions. As seen in the graph in Question 3, Bot 2 performs best at k=10 and Bot 1 performs best at k=20. However, Bot 2 outperforms Bot 1 when comparing their respective "bests". This means that sensing more often with a larger detection square proved to be more efficient. The bot had more information to use at each timestep because of sensing.

Probabilistic Bots

Bot 3

- Initialization
  - Bot 3 initializes the bot and the leak at random open cells in the ship such that they are not the same cell. It initializes the probability of there being a leak in a given cell j for all cells in the ship to be $P(L_j) = \frac{1}{number\ of\ open\ cells - 1}$, except the bot cell has P(L)=0. P(L) is encapsulated in the probLeak attribute for a Cell object.
- While the bot is not the leak:
  - The algorithm goes through all P(L) values among all cells to get the maximum probability. It calculates the shortest path from the bot to that maxProbCell. The bot moves one step in the direction of the shortest path and updates all P(L) values based on stepping into a cell and either finding or not finding the leak. If the bot has made it to the maxProbCell, it will sense and update P(L) accordingly based on if there was a beep.
- Bfs.updateDistances()
  - Modified Dijkstra's Algorithm to update the distances in one traversal. It uses a min priority queue of PQCells, ordering by the distance of each cell from the bot.

- The math for the probabilities of Bot 3 will be explained further in Question 2.

Bot 4

- Bot 4 has the same exact algorithm as Bot 3 except it senses at <u>every other</u> maxProbCell. We implement this by using a boolean variable that inverts its T/F value every time the bot reaches a maxProbCell. By not sensing as often, the number of actions was favorably decreased. This indicates that perhaps we do not need to sense every time we arrive at a maxProbCell like in Bot 3. Bot 4 still utilizes the sense action to give it some inclination as to where the leak might be, but discovering that certain cells are/are not the leak is a more powerful and reliable source of information because it is not dependent on randomness of beeping.

<u>Deterministic Bots - Multiple Leaks</u>

Bot 5

- Initialization
    - Modeled after Bot 1, Bot 5 initializes the bot, leak1, and leak2 such that both leaks are not in the initial detection square and the leaks are not the same cell. All cells in the square set noLeak = true.
- While the bot is not the leak AND (leak1 or leak2 have not been plugged):
    - The algorithm will calculate the BFS shortest path like in Bot 1. It will move the bot along the path. If both leaks have not been plugged:
        - If the bot reached one of the leaks, set the leak that was reached to not be the leak anymore.
        - Regardless, set noLeak = true for the bot and run detSenseAction_MultipleLeaks().
    - Else if the bot reached one of the leaks and the other has been plugged (xor), then return. Then make noLeak = true for the bot and perform a sense in detSenseAction() as in Bot 1. Loop until the second leak is plugged.
- detSenseAction_MultipleLeaks()
    - Similar to detSenseAction(), we get the detection square and check to see if the leak is in the square. If not, set noLeak = true for all cells in the square. If a leak was detected, we only want to look at the cells in the square. We do not want to set noLeak = true for the cells outside the square because there is still another

leak. Instead, we iterate while the bot is not the leak through cells that might potentially be leaks that are within the square. The shortest path is calculated using the overloaded Bfs.detSP_BFS(bot, detSquare) method. The process is similar as in the runBot5() method.

Bot 6

- Bot 6 is modeled after Bot 2 and Bot 5. It senses *at every step*. This allowed Bot 6's smallest number of actions to be less than Bot 5's smallest number of actions for the same reason as Bot 2. Bot 5 performs best at k=10, but Bot 6 performs better at k=20.

Probabilistic Bots - Multiple Leaks

Bot 7

- Bot 7 is modeled after Bot 3 (and Bot 5 to incorporate the logic for multiple leaks). It admittedly calculates the probabilities incorrectly because it calculates them as in Bot 3 without considering the multiple leaks. It initializes the bot, leak1, and leak2 such that they are not the same cells. The P(L) for all cells is initialized as in Bot 3.

Bot 8

- Initialization
    - Bot 8 is modeled after Bot 7 but calculates the probabilities *correctly*. It initializes the bot and the two leaks at random open cells in the ship such that they are not the same cells. The pairings are stored in a hashmap of hashmaps called pairings, which is of type HashMap<Cell, HashMap<Cell, Double>>. Every pair of cells j and k has its own unique probability that those two cells in the pairing have leaks, namely $P(L_j, L_k)$. Bot 8 initializes this probability to be $\frac{2}{n(n-1)}$ where n is the number of open cells - 1 (excluding the bot), except pairings that include the bot which are 0 and not included. The probability that a leak is in a given cell j, $P(L_j)$, is encapsulated in the probLeak attribute for a Cell object. This is calculated to be the sum of all the pairings that include cell j.
    - To get the probability for a pair (Cell j, Cell k), we have pairings.get(j).get(k) or pairings.get(k).get(j) → both correspond to the same value.
    - To set the probability for pairs, we iterate through all open cells and for each cell j, we create a new HashMap<Cell, Double> which will contain the probabilities of the pairs that contain cell j. For all pairs (j, k), where k is an open cell other

than j, when the k cell comes before the j cell then that means that pairing probability has already been calculated before when calculating $P(L_k, L_j)$, so we reuse that probability. For k cells that come after j, we set the probability to be the initial probability.
    - For a cell j, as we set the probabilities for the pairings, we sum them up into a sumProbLeak variable which will become $P(L_j)$ for j. This will be used to determine the maxProbCell to go to.
- While the bot is not the leak and neither leak has been plugged:
    - Just like Bot 3, the algorithm goes through all P(L) values among all cells to get the maximum probability. It calculates the shortest path from the bot to that maxProbCell. The bot moves one step in the direction of the shortest path and updates all P(L) values based on stepping into a cell and either finding or not finding the leak in the updateProb_Step_Bot8() method. If the bot has made it to the maxProbCell, it will sense and update P(L) accordingly based on if there was a beep in the probSenseAction_Bot8() method. Loop until one leak has been plugged.
- Once one leak has been plugged, the procedure will be the same as Bot 3. While the bot is not the leak:
    - The algorithm goes through all P(L) values among all cells to get the maximum probability. It calculates the shortest path from the bot to that maxProbCell. The bot moves one step in the direction of the shortest path and updates all P(L) values based on stepping into a cell and either finding or not finding the leak. If the bot has made it to the maxProbCell, it will sense and update P(L) accordingly based on if there was a beep.
    - The probability updates are the same as for Bot 3 because now there is only one leak.
- probSenseAction_Bot8()
    - Get the beep probability for leak1 and attempt to beep. Do the same for beep2. If either leak caused a beep, call the updateProb_Sense_Bot8() method with a beep probability $P(B \text{ in } i \mid L_j, L_k) = 1 - ((1\text{-prob1}) * (1\text{-prob2}))$. If neither caused the beep, call it with a probability of (1-prob1) * (1-prob2).

- updateProb_Sense_Bot8()
    - Calculate the denominator to be the sum of all the pairings multiplied by the beep probability for that pair. Then create a new HashMap<Cell, HashMap<Cell, Double>> called newPairings and calculate the new pair probabilities to be $\frac{P(L_j,L_k)*P(B\ in\ i \mid L_j,L_k)}{\sum_{j'}\sum_{k'}P(L_{j'},L_{k'})*P(B\ in\ i' \mid L_{j'},L_{k'})}$. After creating newPairings, set pairings = newPairings.
- updateProb_Step_Bot8()
    - Calculate the denominator to be 1-P(L_i) where cell i is the bot cell. Then create a new HashMap<Cell, HashMap<Cell, Double>> called newPairings and calculate the new pair probabilities to be the current pair's probability divided by the calculated denominator. Adjust the P(L_j), which is the sum of the pairs for j, for every cell j accordingly.
- The math for the probabilities of Bot 8 will be explained further in Question 2.

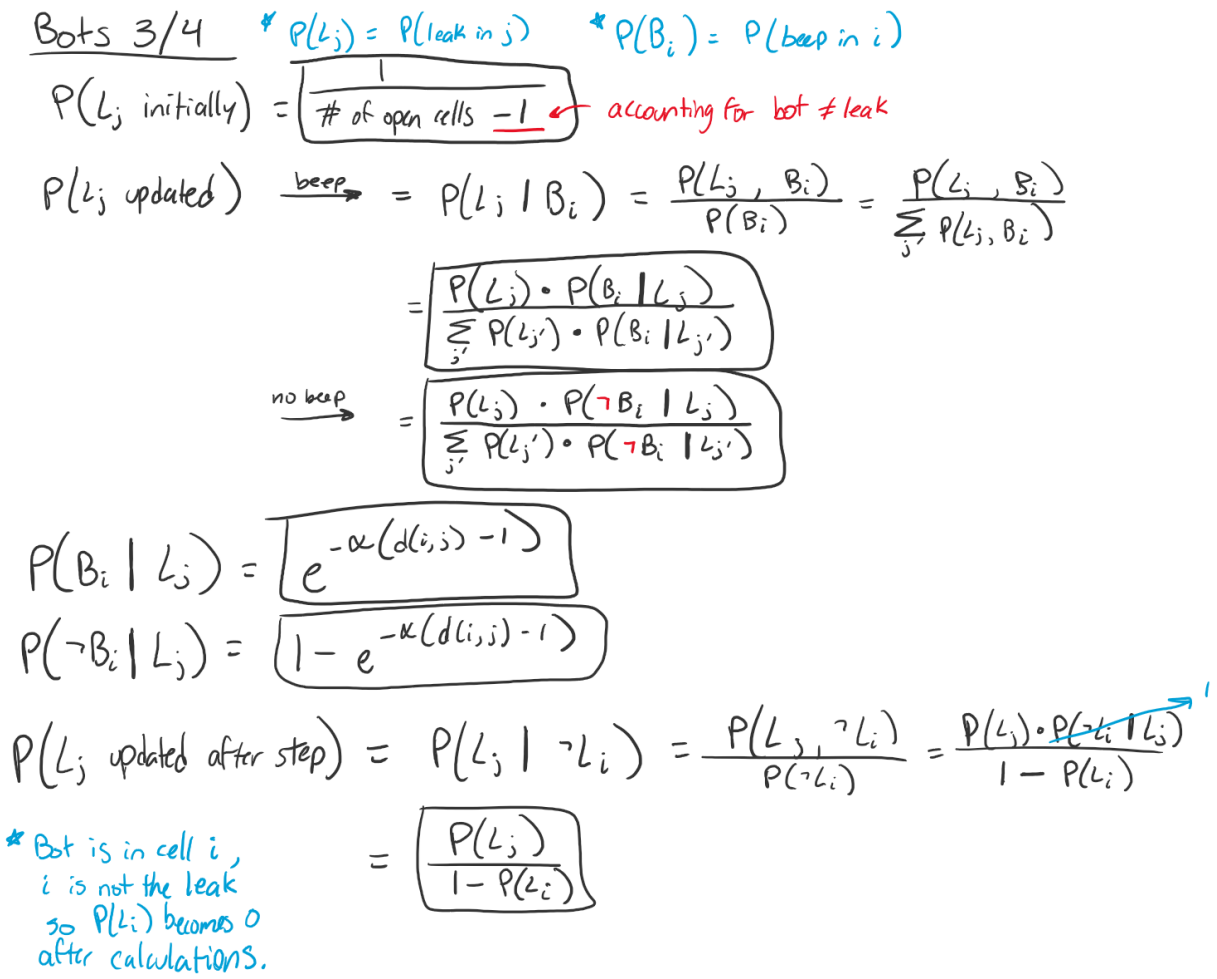
Bot 9

- Bot 9 is modeled after Bots 8 and 4. It essentially has the same algorithm as Bot 8 but senses at every other maxProbCell.

2. How should the probabilities be updated for Bot 3/4? How should the probabilities be updated for Bot 8/9?

- The probabilities for Bots 3 and 4 are updated so that each time the bot senses and/or moves, the probabilities of the opened cells will be updated. If Bot 3 or 4 receives a beep, then the probabilities of the cells closer to the location of the bot would increase, whereas cells further away would decrease. This is due to the fact that you are more likely to receive a beep for Bots 3 and 4 if the leak is close by. This can be assumed by the formula given in the write up: $e^{-\alpha(d-1)}$. Alpha, α, represents the sensitivity of the beeper and d, distance, represents the distance between the bot and the leak. These two variables determine the probability of the bot receiving a beep. If the bot enters a new cell that does not contain a leak, the probability of the cell is set to 0. The other cells are then updated to account for that missing probability. If the bot chooses to sense, the reception of a beep, or lack thereof, would then warrant a probability update where receiving a beep would increase the probability of the cells closer to the bot and not receiving a beep

would decrease the probability of cells closer to the bot. Contrastly, if the bot receives a beep, the probabilities for cells further from the bot would decrease, but if the bot does not receive a beep, the probabilities of the cells further from the bot would increase. No beep indicates that a leak could potentially exist in a cell further from the bot rather than a cell closer to the bot, which is why the combination of increasing further cells' probabilities and decreasing the probability of closer cells containing the leak must occur. The math for updating these probabilities is as follows:



- The probabilities for Bots 8 and 9 get updated similar to Bots 3 and 4 in respect to when probabilities get updated, however, considering the fact that there are now two leaks involved, a little bit more math is needed to account for the different ways a beep could be received. To explain, the code uses a hashmap of hashmaps to keep track of the probability for a particular cell and then within that cell, exists another hashmap that

contains individual probabilities for a particular pairing of cell i, with any other cell j. This hashmap for a particular cell i, would need to be updated every time a sensing a move action occurs. Additionally, the overall hashmap that keeps track of all potential cells, i, would also need to be updated. That is, all cells, i, must have updated pairing probabilities of leaks occurring in cell i and all other cells, j, every time a bot moves or takes a sensing action. In order to calculate what that probability should be for each pairing, as well as other probability calculations, the following math was used:

Bots 8/9 * $P(L_j, L_k)$ = P(leak in j and leak in k)

*After the first leak is plugged, $P(L_j)$ gets updated as in bots 3/4.

$$P(L_j) = \sum_k P(L_j, L_k) \quad \text{where } j \neq k$$ sum of the pairings

$$P(L_j, L_k \text{ initially}) = \frac{2}{n(n-1)}$$ where n = # of open cells - 1

$P(L_j, L_k$ updated) beep →

$$\frac{P(L_j, L_k) \cdot P(B_i \mid L_j, L_k)}{\sum_{j'} \sum_{k'} P(L_{j'}, L_{k'}) \cdot P(B_i \mid L_{j'}, L_{k'})}$$ s.t. a pair $(j,k) = (k,j)$ and is not counted twice

no beep →

$$\frac{P(L_j, L_k) \cdot P(\neg B_i \mid L_j, L_k)}{\sum_{j'} \sum_{k'} P(L_{j'}, L_{k'}) \cdot P(\neg B_i \mid L_{j'}, L_{k'})}$$

$$P(B_i \mid L_j, L_k) = 1 - P(\text{neither leak cause } B_i \mid L_j, L_k)$$
$$= 1 - P(j \text{ causes } \neg B_i \mid L_j) \cdot P(k \text{ causes } \neg B_i \mid L_k)$$
$$= 1 - [1 - P(B_i \mid L_j)] \cdot [1 - P(B_i \mid L_k)]$$ calculated from Bots 3/4 to be $e^{-\alpha(d-1)}$

$$P(\neg B_i \mid L_j, L_k) = [1 - P(B_i \mid L_j)] \cdot [1 - P(B_i \mid L_k)]$$

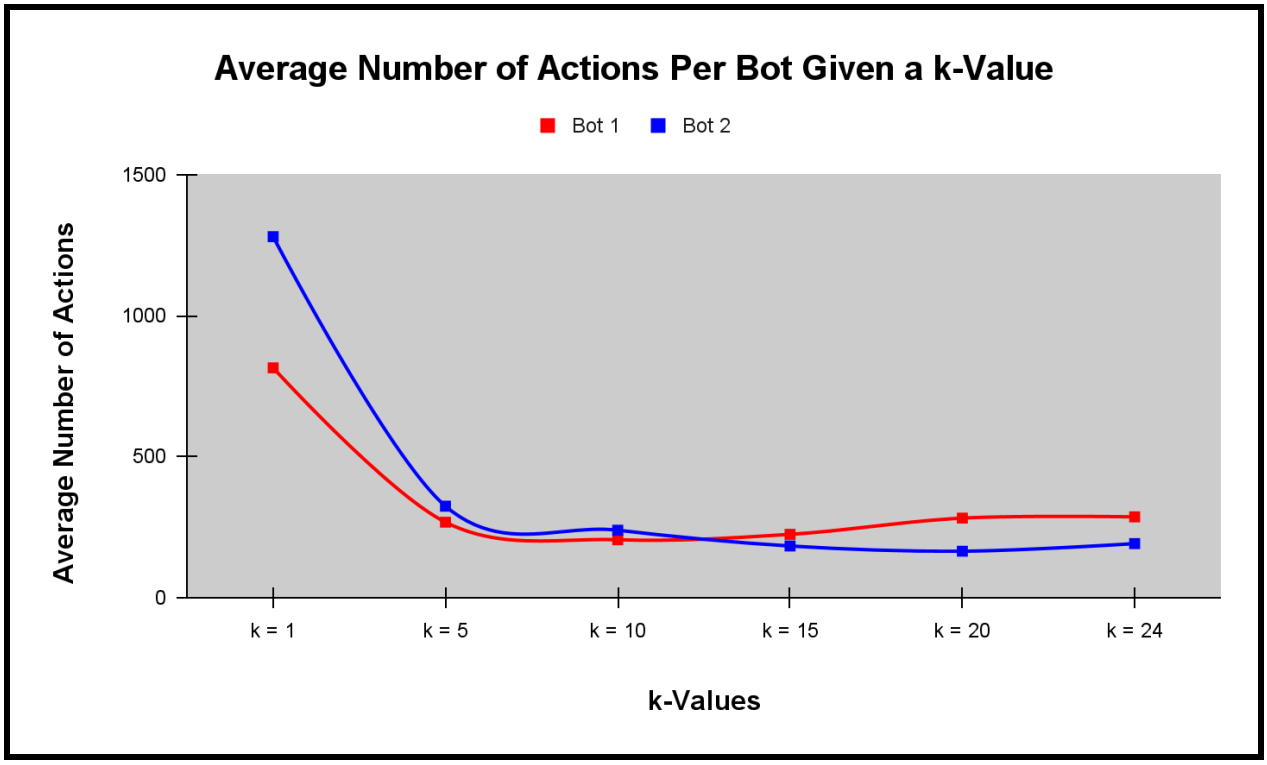
$P(L_j, L_k$ updated after step) $= \frac{P(L_j, L_k)}{1 - P(L_i)}$

* Bot is in cell i, i is not the leak so $P(L_i)$ becomes 0 after calculations.

3. Generate test environments to evaluate and compare the performances of your bots. Your measure of performance here is the average number of actions (moves + sensing) needed to plug the leak.

- **Note:**
    - For the purposes of analyzing the efficiency of each bot most accurately between bots 1-9, we decided to omit cases where the bot spawns on a leak from the test results at t = 0.
    - Bots 1-7 were run on a 50x50 ship with 100 test cases for each k or **α** value.
    - Bots 8 and 9 were run on a 30x30 ship instead (still with 100 tests for each **α**) due to taking a very long time for each test for a larger size ship.
- **Bot 1 vs Bot 2, as a function of k.**

k-values used for testing	Bot 1	Bot 2
k = 1	814.65	1279.6
k = 5	267.58	324.42
k = 10	205.4	239.14
k = 15	224.5	183.18
k = 20	185.48	185.48.72
k = 24	286.5	191.68

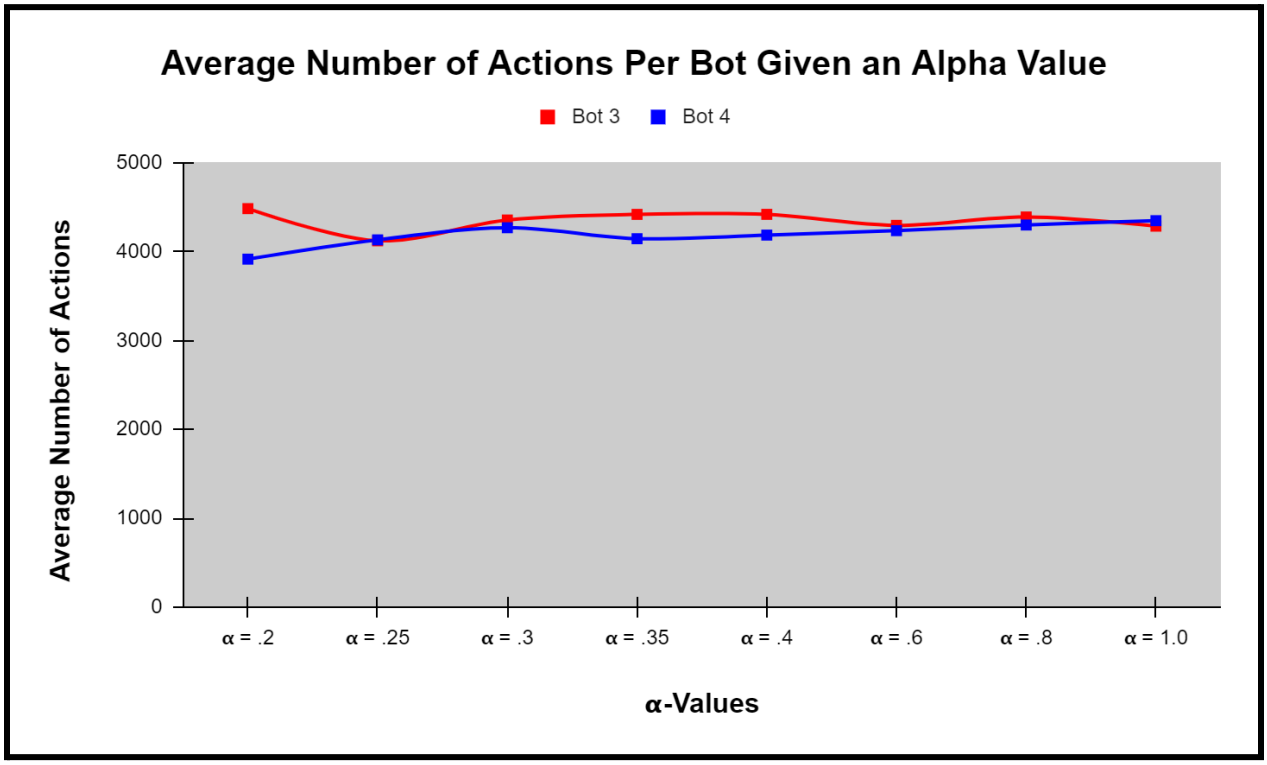


- **Bots 1-2 Summary**
    - The results produced by Bot 1 indicates that there may be an optimal k-value at k = 10. For Bot 2, that optimal k-value appears to be at k = 20. For lower values of k, Bot 1 seemingly does better than Bot 2. This can be explained by the fact that Bot 2 could have the potential to overshoot the leak with a smaller detection square, meaning that it could be difficult for Bot 2 to detect a leak with a small detection square if its movements are more drastic than Bot 1. For higher values of k, Bot 2 does better. This is due to the fact that Bot 2 would be able to sense a larger area of the ship quicker than Bot 1.
- It is possible to conclude that there is an optimal detection square of size (2k+1)x(2k+1) that allows the bot to find the leak in an optimal amount of actions. This optimized k-value occurs at k=blank for Bot 1. The optimal k-value occurs at k=blank for Bot 2.

- **Bot 3 vs Bot 4, as a function of *α*.**

	Bot 3	Bot 4

α = .2	4482.33	3914.03
α = .25	4120.2	4130.99
α = .3	4354.28	4267.32
α = .35	4417.46	4143.89
α = .4	4417.8	4184.51
α = .6	4294	4234.46
α = .8	4388.51	4298.0
α = 1.0	4285.04	4346.84

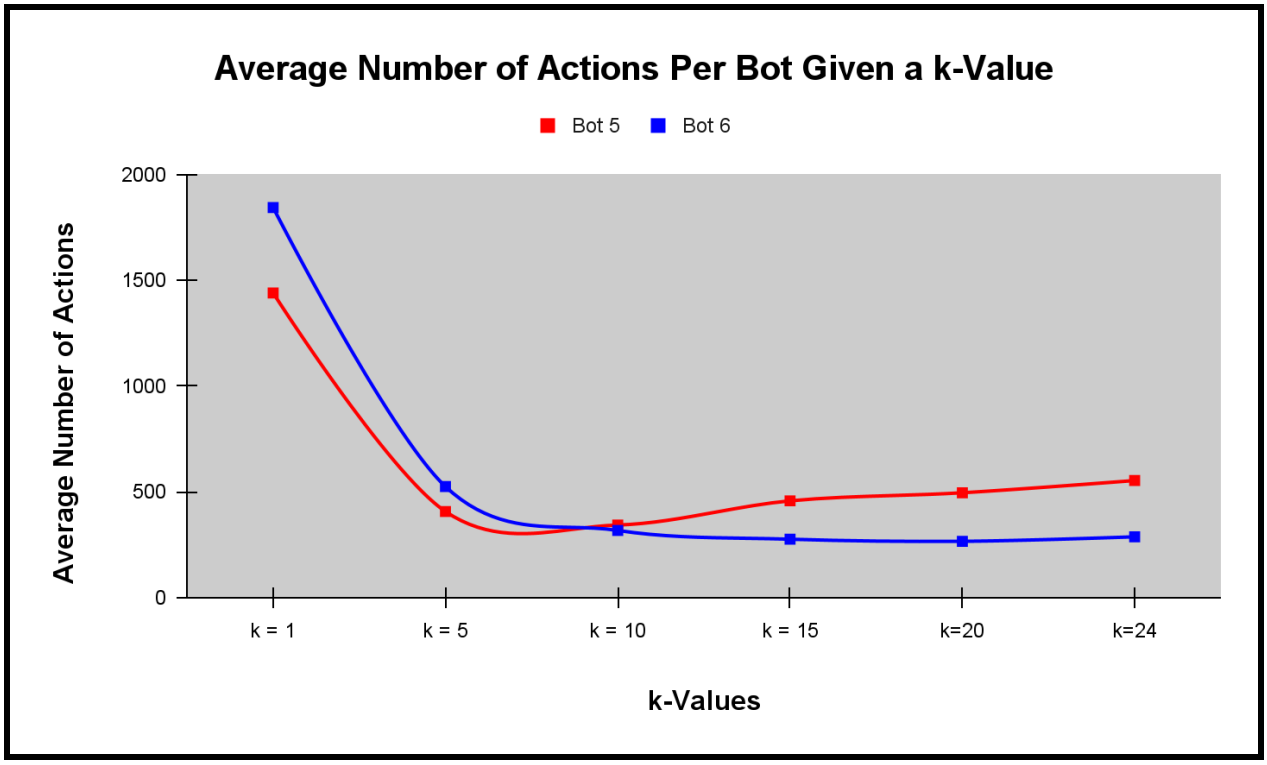


- **Bots 3-4 Summary**
    - The results found for Bots 3 and 4 were in line with what we thought the results would produce, in terms of how well the Bots would perform against each other. Almost consistently, Bot 4 does better than Bot 3. This can be attributed to the fact that Bot 4 takes less sense actions to locate where the leak is. This produces a less number of actions and provides more weight to each sense action. That is, instead of having a bunch of probabilities that have the chance to be buffered by a

series of sense actions that could return either beep or no beep, just sense once for every other maxProbCell encountered while searching for the leak. This proved to help increase the efficiency of Bot 4 and reduce the number of actions taken by the bot to find the leak. Bot 4 takes probability into account, similar to Bot 3, but rather than constantly sensing and leaving receiving a beep or not up to a bunch of runs for probability, the Bot will sense once every so often to locate the next cell that could potentially have a leak, or the next maxProbCell. Bot 3 senses after every step which would, understandably, increase the number of actions Bot 3 will take, regardless of how accurate the sensor may be.

- **Bot 5 vs Bot 6, as a function of k.**

	Bot 5	Bot 6
k = 1	1440.09	1843.83
k = 5	407.26	525.08
k = 10	343.45	317.81
k = 15	457.73	276.38
k=20	495.85	266.44
k=24	553.7	287.78

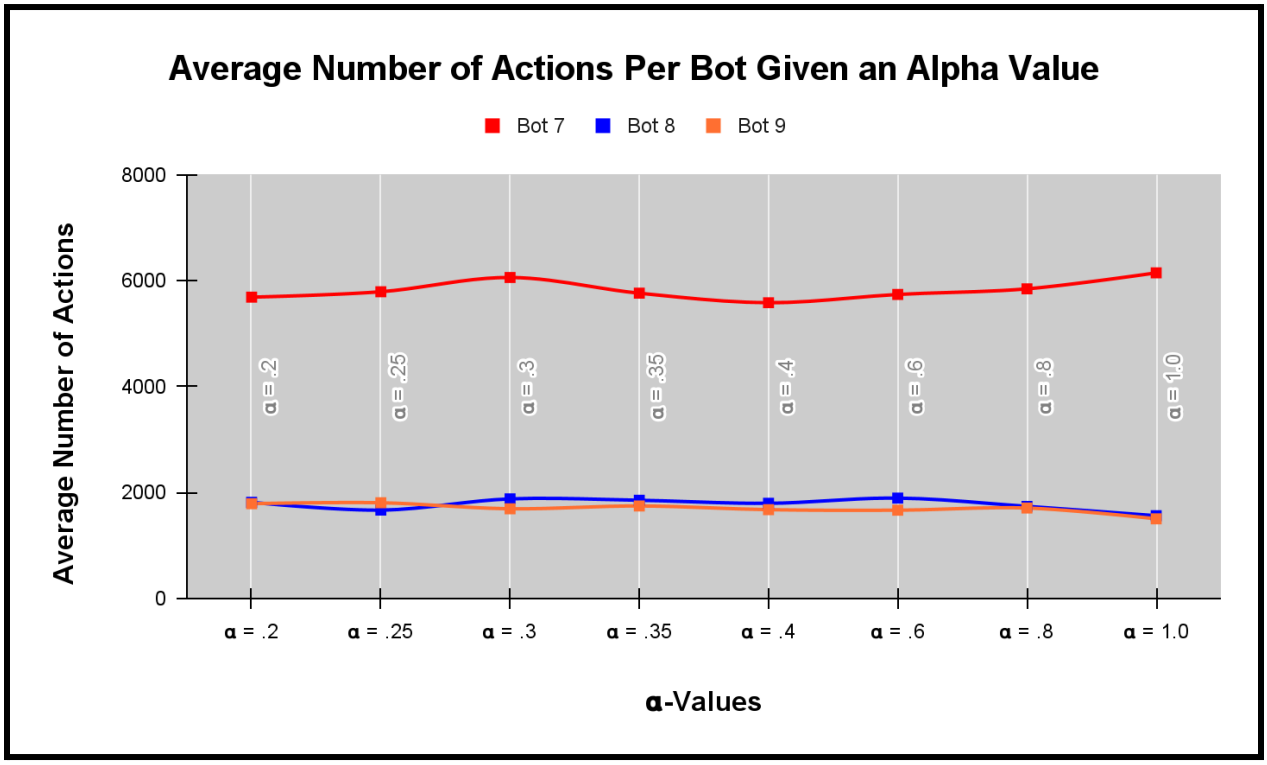


- Bots 5-6 Summary
    - Not to our surprise, Bot 6 did better than Bot 5 when handling multiple leaks at higher values of k. This can be understood when taking into consideration that Bot 5 runs code similar to that of Bot 1, just taking another leak into account. Similarly, Bot 6 runs code similar to Bot 2 while also taking the multiple leaks into consideration. Since Bot 2 generally did better than Bot 1, it could be anticipated that Bot 6 would, on average, find the leaks in a less number of actions than Bot 5. Optimal value of k found for Bot 5 is k = 10 and for Bot 6 it is k = 20.

- **Bot 7 vs Bot 8 vs Bot 9, as a function of α.**

	Bot 7	Bot 8	Bot 9
α = .2	5688.45	1815.75	1789.67
α = .25	5790.42	1667.85	1806.7
α = .3	6059.93	1880.1	1693.58

α = .35	5763	1854.17	1746.69
α = .4	5583.5	1796.4	1676.65
α = .6	5739.4	1895.23	1666.01
α = .8	5845.4	1739.2	1707.62
α = 1.0	6147.17	1562.11	1504.21



- Bots 7-9 Summary
    - Bot 7 does significantly worse than Bots 8 and 9, though that is by design. Bot 8 and 9 are very similar in efficiency, however, for almost every alpha value, Bot 9 does slightly better. When taking the probabilities of the two beeps into account, the probability of receiving a beep would be increased for both bots. This forces the question: Does sensing less turn out to be beneficial for the bot? Previously, for Bot 4, it seemed that sensing less would turn out to be beneficial in that less actions were taken to find the leak. Similarly, when using that logic for Bot 9, it was found that for most alpha values, Bot 9 would do slightly better than Bot 8, which sensed after every step to receive a beep (or not). It seems that for multiple

leaks, sensing or not sensing has the same effect. That is, Bots 8 and 9 are extremely comparable in number of actions, even though Bot 8 does a very high number of action senses, whereas, Bot 9 does a very low number of senses, but still returns a total number of actions very close to that of Bot 8.

4. Speculate on how you might construct the ideal bot. What information would it use, what information would it compute, and how?

- When thinking about this project and the requirements for an ideal bot, it is impossible to not consider how a better beeper scanner could be constructed. If there was a mathematical formula for, or rather, an indicator as to how far a leak is depending on a beep, it would significantly increase the accuracy of the updated probabilities. Perhaps the beeper returns a faint beep to indicate that the leak is further away and a stronger, or louder, beep to signify a leak that is close. To calculate something like this, the distance of the leak to the bot must be known and calculated, which would help with determination of strength of beep. Furthermore, we could add a modification to the beeper so that if there were multiple leaks on the ship, the scanner could return two beeps. Of course, the beep logistics would be similar to that previously mentioned. That is, it is possible for a bot to receive 2 strong beeps, a strong and a fainter beep, or two beeps that are faint. In each of these cases, the 2 strong beeps should increase the probability of a leak being closer to the bot, and therefore, update those cells. If you had a strong beep and a really faint beep, the bot would update the probabilities so that cells close to the bot and cells really far away from the bot would have an increased probability, however, the cells that would be in between the two theoretical leaks would have probabilities that would decrease. In the last case, the bot receives 2 faint beeps, the probabilities would update so that the further cells would have an increase in the probabilities, meanwhile the cells that are closest to the bot would have a probability update that decreases. Being able to have a beep that could have sound decibels that are proportional to the distance between the bot and the leak, the bot would be able to have a very good estimate as to where the leaks are. This would, in turn, decrease the amount of actions the bot would need to take significantly. Furthermore, the data obtained from sensing would become more useful to the bot, thus also decreasing the amount of senses

the bot would need to complete for a strong certainty of the location of the cell. After plugging the first leak, the bot should sense again for a beep. This time, the bot only hears one beep as there is only one leak left. The bot would be quite certain as to where the leak may be given how loud a beep is. The bot will update probabilities for the singular leak and start its traverse to the cell with the highest probability.

- Another possibility for modifications for an ideal bot could be adding simulations to the bot before the bot moves. This would give the bot an "inclination" as to where the cell with the highest probability of having a leak may be. Within these simulations, multiple sensing actions could occur, the sensor being the sensor previously outlined, so that the bot could make the most informed decision as to where on the ship a leak or leaks may be. If you sense multiple times, theoretically, you should get a beep of similar decibels each time, assuming the bot is not moving. Therefore, if a beep of similar decibels is heard multiple times, the bot can get a general idea as to how far the leak probably is and can update the probabilities of the cells accordingly. After running simulations, the bot can choose a path to go down, sensing every so often to make sure it does not overshoot the leak as the cell with the highest probability might not exactly be the cell with the leak, but is within a few cells of where a leak could be. This run simulation can be conducted for an arbitrary amount of time and for either a singular or multiple leaks. If more multiple leaks, the bot should choose to travel in a direction towards the louder leak, or by indication, the closer leak.

***Note: Both Patryk and Mathew worked very closely together on each aspect of the project so it is hard to say who contributed what specifically.**