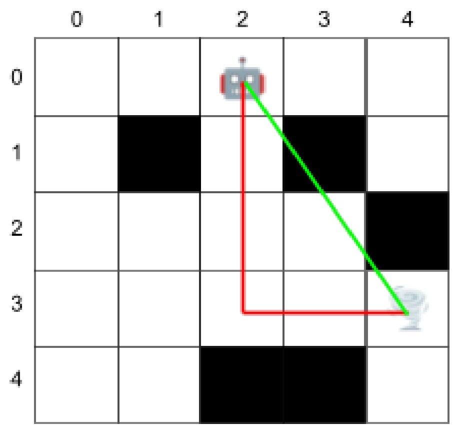
Nikhil Agarwal
November 15, 2023
01:198:440 - Dr. Cowan
Project 2 Writeup (Solo) - Airing Out

Bot Design

I developed solutions to this project using OOP in python. I have created Ship objects for each group of bot (grouping by k, alpha, and number of leaks). main.py contains all of the test functions that will automatically generate test environments and run experiments. data.py holds all of the data that I generated with the test functions.

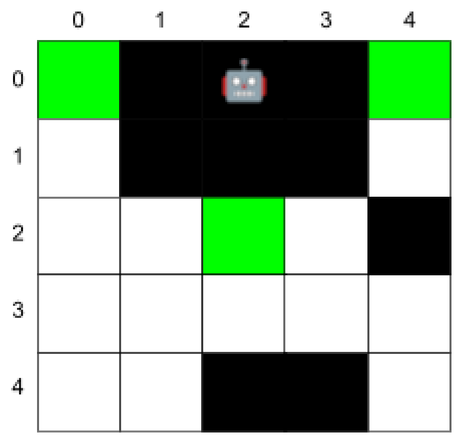
Bot 1 -
This bot has three main functions: scanning the surrounding cells for a leak, choosing the next cell to move to, and path planning. Based on the given rules for this robot, it must move to the *closest* cell that could potentially contain the leak scan for the leak. If the scan *does not* detect a leak, repeat for the next closest cell. If however, the scan *does* detect a leak, visit each cell that has not yet been visited within the range of the scan.

The closest cell is calculated using the Euclidian distance between cells on a grid. Since I use a



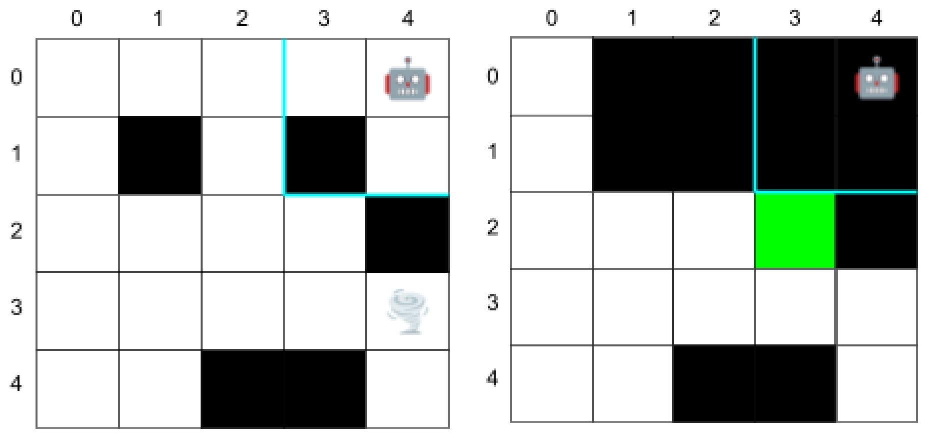
two-dimensional array to store the contents of the ship, each cell on the ship has an associated coordinate point. Take the diagram of a sample 5x5 ship, with k = 1, on the left, where the black cells represent walls. The bot is located at row index 0 and column index 2, while the leak is located at (3,4). Therefore the distance that my robot would calculate from itself to the leak would be approximately $\sqrt{3^2 + 4^2} \approx 3.61$.

Now that bot 1 can correctly calculate the distance from itself to another cell, we need a way of letting the bot know which cells it can choose and which it can't. For this, I maintain a separate two-dimensional array that acts as the bot's memory. For the ship above, the memory of the bot would look something like the figure on the right, where black indicates *impossible for a leak*, and white indicates *possible for a leak*. Since we automatically place the leak outside the initial range of the bot's scanners, the bot can automatically block all cells within its initial scan range as well as any cells that have walls. In my implementation, I

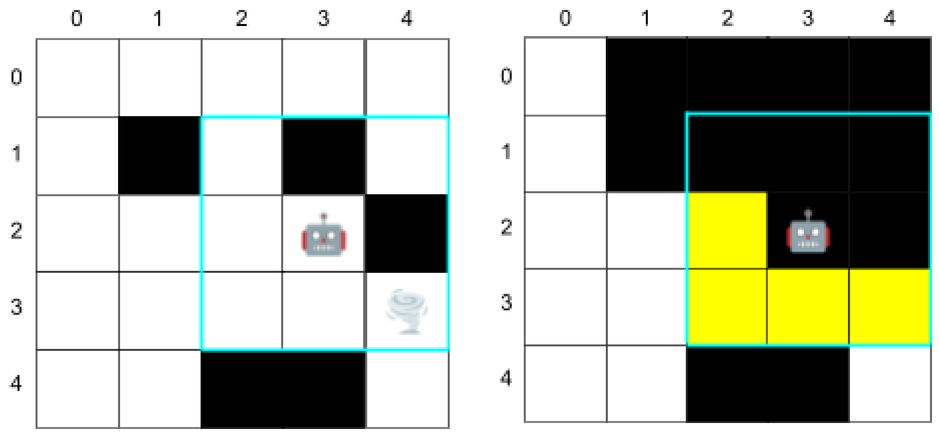


maintain a *set* of *tuple* objects for all open cells (cells that may contain the leak). By looping through this set, and calculating the distance from the bot to every possible leak location, the bot will find that there are three cells that are the closest, and randomly choose one to travel to. It's important to note that the leak itself is not displayed in the bot's memory, because it *doesn't* know where the leak is yet.

After the bot determines the cell that could possibly contain the leak, it uses an A* algorithm with a Euclidian distance heuristic to calculate the optimal path from its current location to its chosen target cell. Once the bot enters the target cell, it checks to see if there is a leak. If there is a leak, then the task is done. However, if there is no leak, the bot uses its scan function. In my implementation, I loop through each cell that is in the *possible leak* set AND also within the range of my scanners.

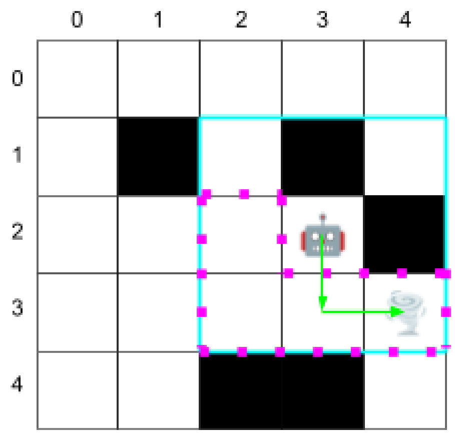


In the diagrams above, the bot has moved to cell (0,4). The bounds of the bot's scanners given k = 1 are drawn as well. Since the leak is not within the range of the scanners, bot 1 marks all cells within the scan range as *impossible* and will choose the next closest *possible* cell (2,3).

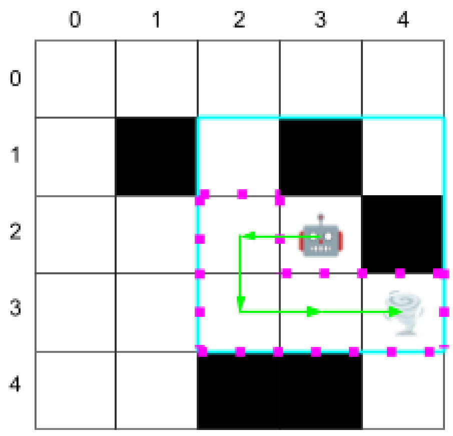


After moving to cell (2,3), the bot will automatically mark its location as *impossible* since the leak is not at that location, and then it will scan. As shown, the leak is within the range of the scanner and will therefore update its memory to show that the leak has been detected. The bot will then travel to each cell with a known possible leak (marked in yellow) and check for a leak. In my implementation, the bot uses two while loops to execute this task. The first while loop executes until it has detected the leak through scanning, at which point the bot will transition into the second while loop which will loop until the leak is found. If the leak is found at any point simply by moving to a cell, the loops terminate. Throughout, the bot keeps track of the number of moves it makes using a counter variable. Each move made by the bot increments the counter by 1 as well as each scan.

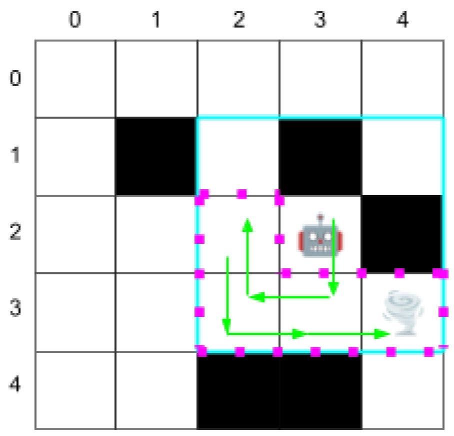
After detecting the leak, my bot can take any number of paths to the leak depending on luck. On the example ship for example the bot can take the following paths. The bounds of the scan and the bounds of the cells that were detected to potentially contain a leak are shown



Takes two time steps



Takes four time steps

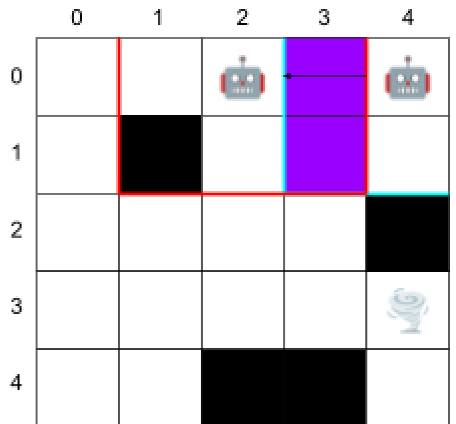


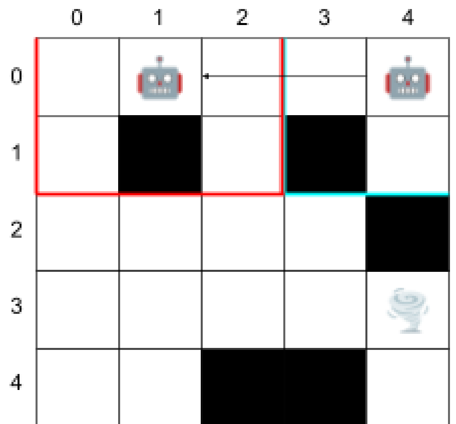
Takes six time steps

Bot 2 -
This bot is different from bot 1 because it uses different logic to choose the next cell to move to. In bot 1, we simply moved the closest cell that could possibly have the leak. However, the downside of bot 1 is that it does not efficiently use it's scanner. Ideally, the bot should move to a state that will allow it to scan the most number of unscanned cells in one timestep. Therefore, bot 2 employes a local search algorithm, where the next cell that will be visited, will be a cell that has the most number of unscanned possible cells within the scanners 2k+1 x 2k+1 square grid.

In experimenting for solutions, I developed two different algorithms that yielded promising results. I call these bot2A and bot2B. The final version of bot 2 is an fusion of the two variants.

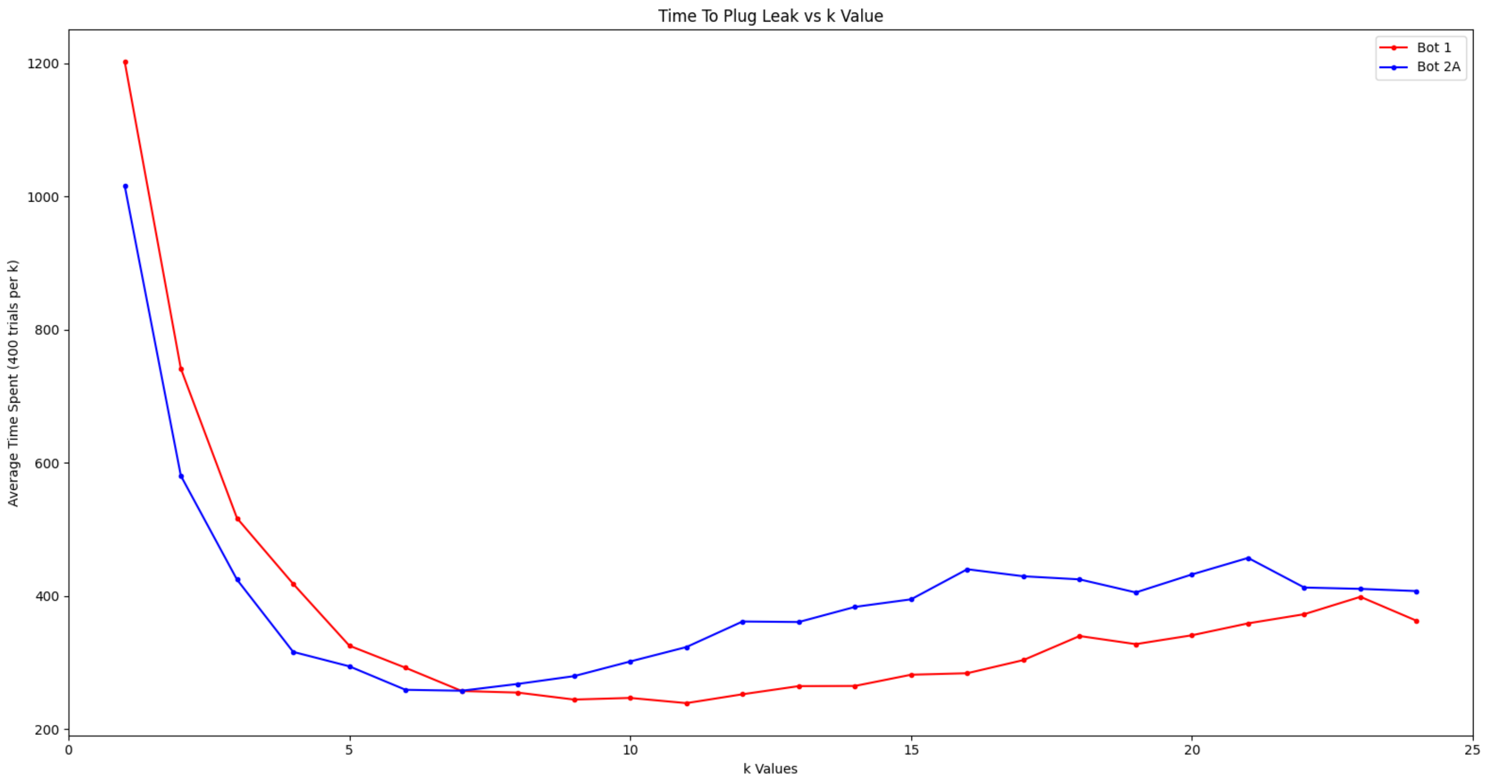
Bot 2A -
This bot will first seek to move either up, left, right, or down by 2k+1 grids. This distance away from the current location of the bot will ensure that there is *no overlap* between cells scanned in the current time-step and the next time-step.
Below, we see that there is an overlap of 2 cells when we choose the next cell with bot1 logic (again k = 1). For the Digram on the right however, there is no overlap.



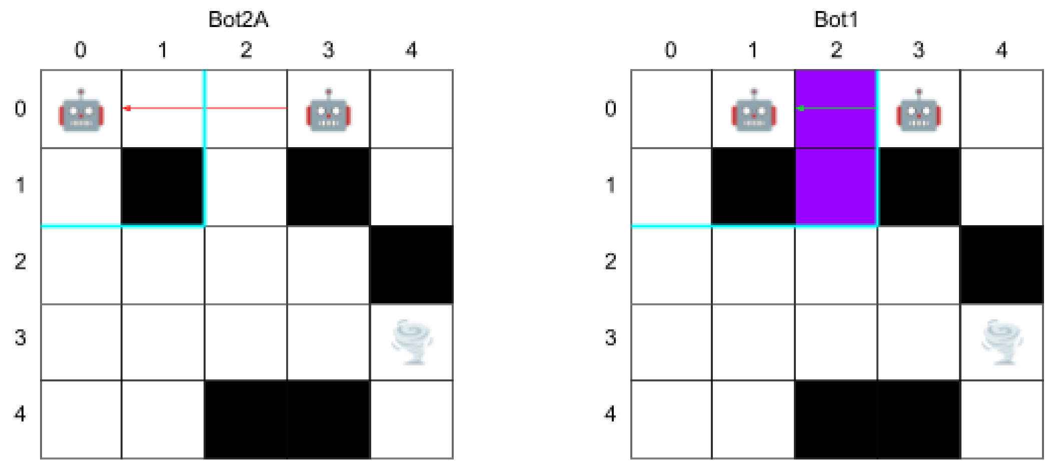


If moving bot2 up, down, left, or right 2k+1 grids, places it outside the ship, the bot will automatically correct itself to a cell in the same direction, on the border of the ship. To determine which direction the bot itself to should travel, my bot simply counts the number of unscanned cells within the range of its scanner if it were to move to a particular location. Bot2A (and Bot2B) maximize this value when choosing the next cell. In the event that all cells 2k+1 cells away in all 4 directions result in the same, a random direction is chosen. Similarly if all 4 directions result in a total of 0 unscanned cells (all cells within the bots immediate state and next states have been scanned), for just that time-step, bot2A will revert to choosing the next cell like bot 1. It's also important to note although Bot2A assume an ideal 2k+1 moves to cells in all directions, when it *actually moves* to those cells, it calculates a path through the ship using an A* algorithm.

Below is a graph of bot2A's performance in comparison to Bot1 (50x50 ship | 400 trails per k value).



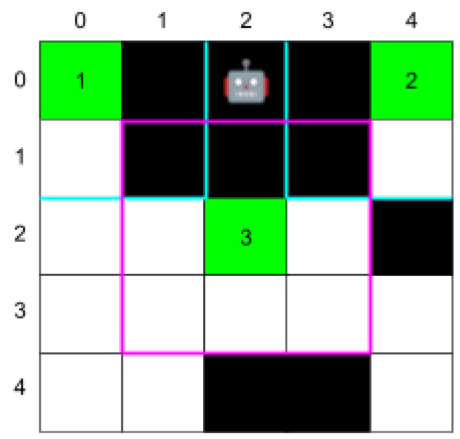
The performance of bot2A is *significantly* better than Bot1 but only when k < 7. Otherwise, bot2A actually performs quite poorly compared to bot1. When the size of the sensor is low, maximizing the number of unscanned cells in this relatively simply way of simply moving up, down, left, or right is effective. However, when the scanners scope increases, It becomes just as wasteful to waste scanning space on cells that are not on the ship, as it is wasteful to scan cells that we *know for sure* were scanned in the previous time-step. Take the diagrams below (k = 1).



The figure directly above, shows which cell bot2A would choose given that It chooses to move left. However, since the bot is at the border, it does not scan any new cells compare o bot1. Even though there is an overlap of scanned cells in adjacent time-steps in bot1, it does not waste an extra move where bot2A would. As scanner sizes get very large, bot2A actually wastes a lot of time moving the edges of the ship when it's not necessary. To fix this, I tried a second implementation.

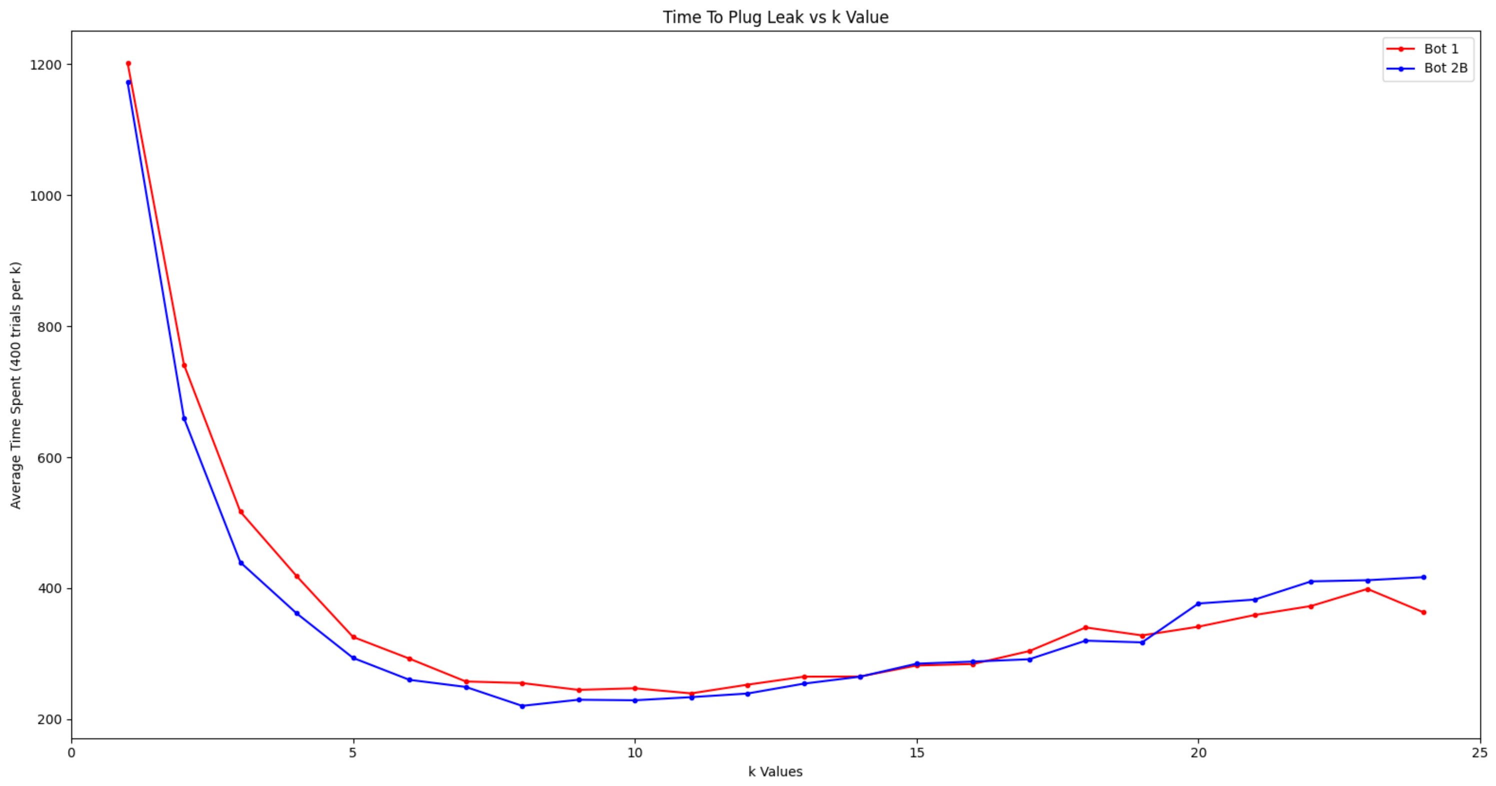
Bot 2B -

For this bot, I aimed to minimize the distance the bot would have to move to find a "good" or "better" state. Bot2B searches for the closest cell that could possibly contain the leak, and will *also maximize* the number of unscanned cells

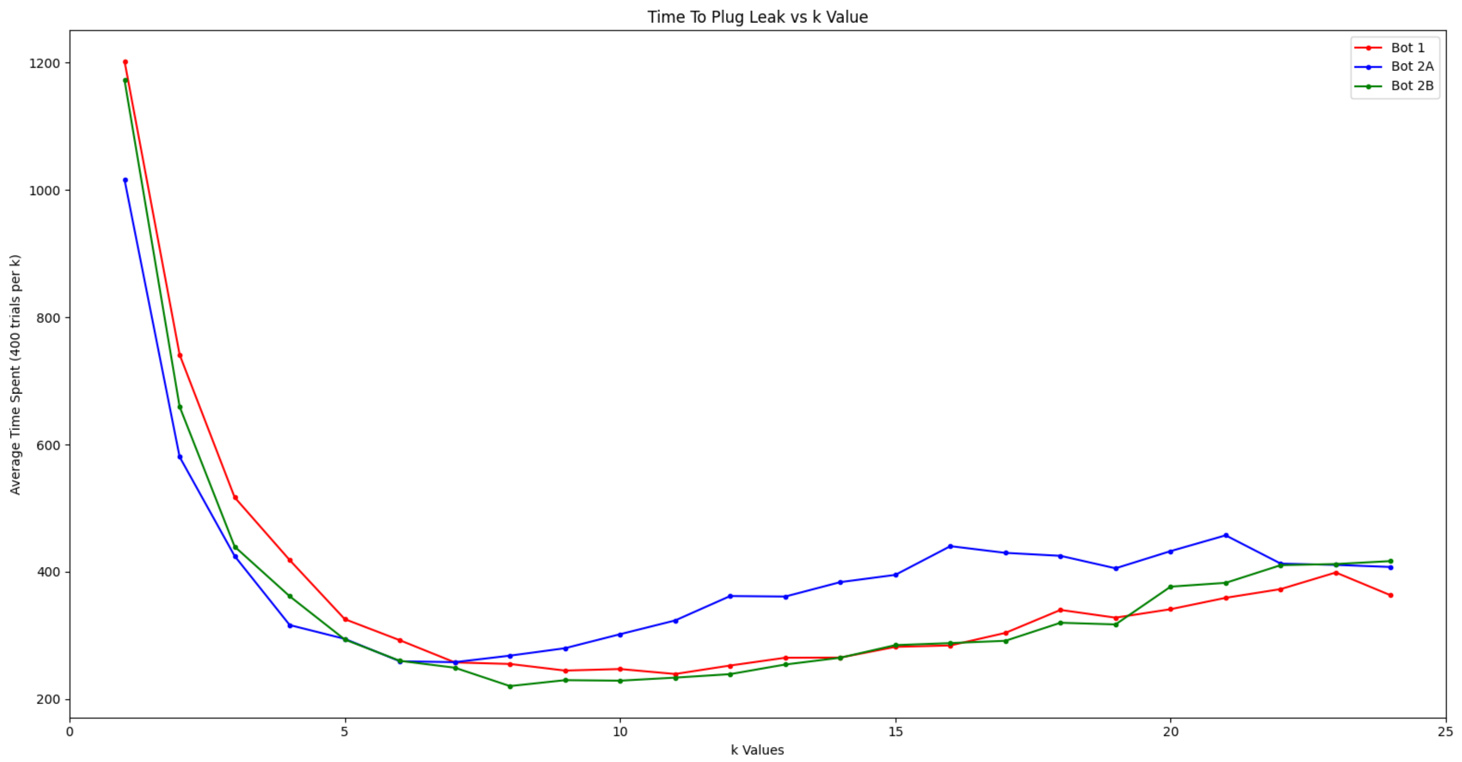


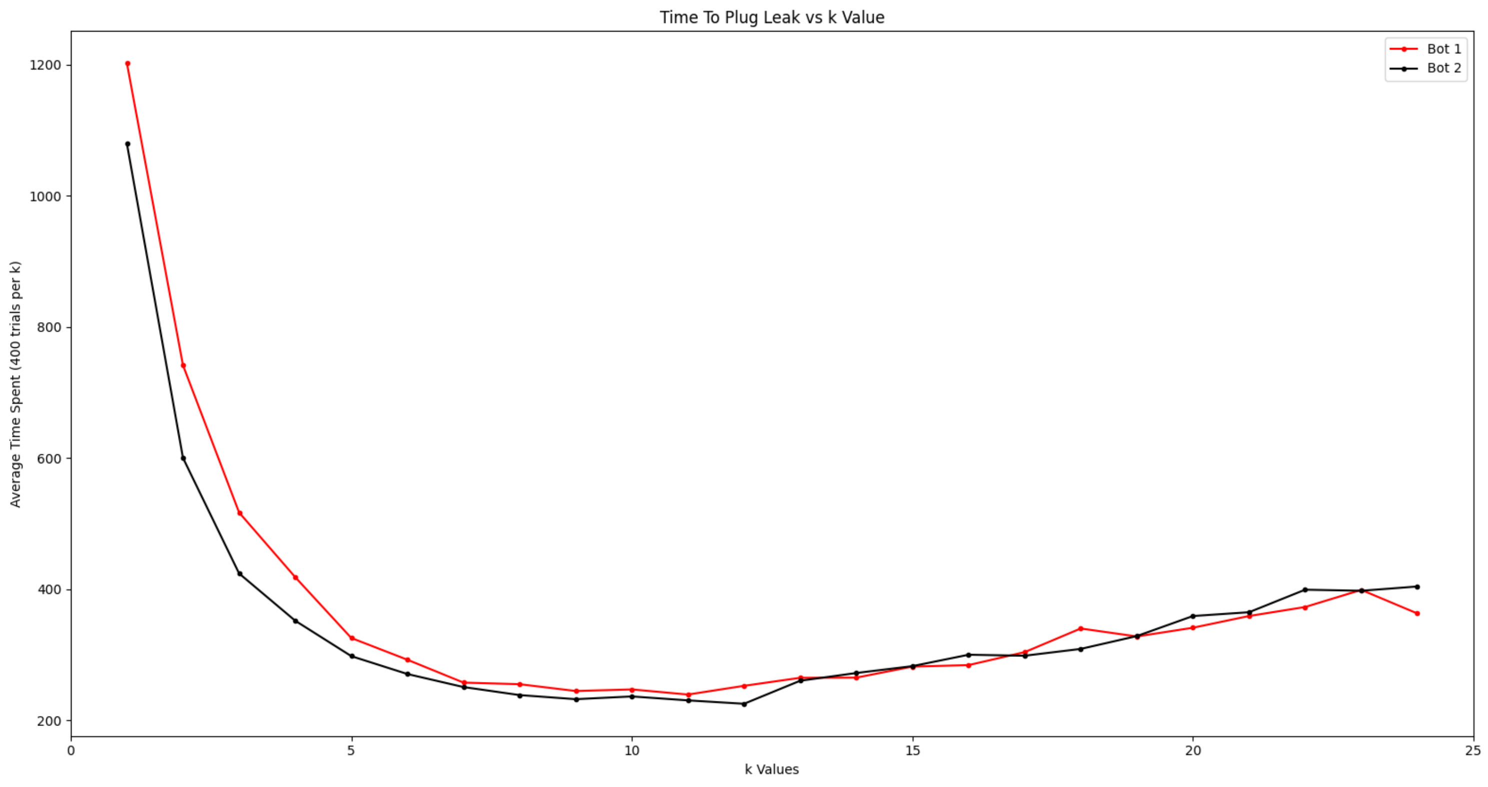
Similar to bot1, bots 2A and 2B both have memory arrays. To the left is an example memory array of a bot with k = 1. The bot can move to any of the green squares. However, cell 3 will allow bot 2B to scan five unknown cells, while cell 1 & 2 will limit the bot to scanning one unknown cell each. Therefore, to give itself the highest probability of detecteing a leak, bot2B will choose cell 3. Path planning is determined by an A* algorithm. Again, in the event of a tie, a cell is randomly chosen.

Below is a grah of Bot2B's performance to Bot1 (50x50 ship | 400 trials per k value).

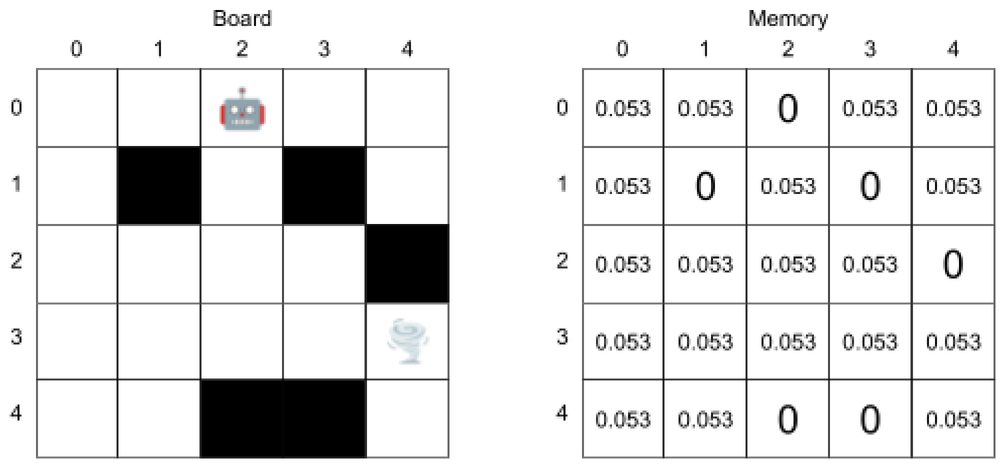


Bot2B also performs better than bot1, but this time, for low k values, as well as k values into the upper teens. Even though bot2B still performs better than bot1 for low k values, I decided that my final version of bot2 would be a fusion of bot2A and bot2B. Bot2A performs *exceedingly well* for k < 7. So for k values less than 7, bot2A logic is used to determine the next cell. Otherwise, bot2B logic is used to determine the next cell. Bot2 triggers a scan as soon after it reaches its goal cell and checks the cell for a leak. Below is a graph comparing Bot1, bot2A, and Bot2B, as well as a graph of the final version of Bot2 compared with Bot1.





Bot 3 -
Bot3 is different than the first two bots in implementation because the behaviour for bot3 does not change during its runtime. Bot3, calculates the probability that each cell on the ship will have a leak, chooses the cell with the highest probability in the *entire ship*, plans a path to that cell with an A* algorithm, checks all the cells along that path for a leak while updating probabilities accordingly, then finally uses it's sensor and updates probabilities in its memory depending on a *beep* or *no beep*. Similar to bots 1 and 2, however, bot 3 keeps a separate memory matrix which holds the probability value for each cell to have the leak.



	0	1	2	3	4
0	0.053	0.053	0	0.053	0.053
1	0.053	0	0.053	0	0.053
2	0.053	0.053	0.053	0.053	0
3	0.053	0.053	0.053	0.053	0.053
4	0.053	0.053	0	0	0.053

When the ship is first initialized, all cells with walls have a 0 probability of having the leak as well as the bot's initial location. Further, every other open cell has an equal probability of containing the leak. Intuitively this makes sense since, the bot doesn't know where the leak is an it only knows where the open cells are, all of which *could* have the leak. I calculate this probability where every cell j (if j is open) has an initial probability of $\frac{1}{number\ of\ open\ cells}$.

To update probabilities there are four cases to consider (A is the current cell of the bot):

1. Update probabilities given there no leak in A
2. Update probabilities given there is a leak in A
3. Update probabilities given *beep heard* while in A
4. Update probabilities given *no beep heard* while in A

We also need some equations of conditional probability:

- $P(X \mid Y) = \frac{P(X \cap Y)}{P(Y)}$
- $P(X \cap Y) = P(X) \cdot P(Y \mid X)$

Case 1 -
We make use of the properties of conditional probability
For each each cell J that can contain the leak (the probability of J is not 0 *and* J != A), the probability of cell J should be updated as:

$$P(Leak\ in\ J \mid No\ leak\ in\ A)$$
$$= \frac{P(Leak\ in\ J \cap No\ leak\ in\ A)}{P(No\ leak\ in\ A)}$$
$$= \frac{P(Leak\ in\ J) \cdot P(No\ leak\ in\ A \mid Leak\ in\ J)}{P(No\ leak\ in\ A)}$$
$$= \frac{P(Leak\ in\ J)}{1 - P(Leak\ in\ A)}$$

$P(Leak\ in\ J)$ is the current value in J and $P(Leak\ in\ A)$ is the value currently in A.
For cell A, the probability becomes 0 because we did not find the leak.

Case 2 -
For this case we actually don't need to update probabilities, we simply terminate our program and output the number of time-steps it took to find the leak.

Case 3 -
We make use of the properties of conditional probability
For each cell J in the ship that can contain the leak (the probability of J is not 0) we update the probability of cell J as:

$$P(Leak\ in\ J \mid Beep\ in\ A)$$
$$= \frac{P(Leak\ in\ J \cap Beep\ in\ A)}{P(Beep\ in\ A)}$$

$$= \frac{P(Leak\ in\ J) \cdot P(Beep\ in\ A\ |\ Leak\ in\ J)}{P(Beep\ in\ A)}$$

$$= \frac{P(Leak\ in\ J) \cdot e^{-\alpha(d-1)}}{\sum_{all\ cells\ K} P(Leak\ in\ K) \cdot P(Beep\ in\ A\ |\ Leak\ in\ K)}$$, d is the length from A→J

$$= \frac{P(Leak\ in\ J) \cdot e^{-\alpha(d1-1)}}{\sum_{all\ cells\ K} P(Leak\ in\ K) \cdot e^{-\alpha(d2-1)}}$$, d1 = len(A→J) and d2 = len(A→K)

$P(Leak\ in\ J)$ is the value currently in cell J, $P(Leak\ in\ K)$ is the value currently in cell K, α is the given value we pass to the sensor.

To calculate the distance from cell A→K, I need an efficient way to calculate the distance from cell A to every cell K in the ship. Bot3 accomplishes this with a BFS transversal over the entirety of the ship. Everytime, a new cell is searched it adds the coordinate and it's distance from A in a map (dictionary object). When calculating $P(Leak\ in\ J\ |\ Beep\ in\ A)$ it then becomes easy to fetch the distance from cell A to any cell in ship for d1 and d2.

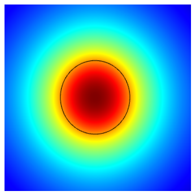
Case 4 -
The logic is the same as Case 3, except we want the probability of *not* receiving a beep. Therefore, for each cell J in the ship that could contain the leak, we update its value to be:

$$\frac{P(Leak\ in\ J) \cdot [1-e^{-\alpha(d1-1)}]}{\sum_{all\ cells\ K} P(Leak\ in\ K) \cdot [1-e^{-\alpha(d2-1)}]}$$, d1 = len(A→J) and d2 = len(A→K)

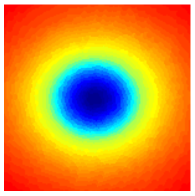
$P(Leak\ in\ J)$ is the value currently in cell J, $P(Leak\ in\ K)$ is the value currently in cell K, α is the given value we pass to the sensor.

Additionally in my implementation, bot3 calculates the location with the maximum probability *while* updating it's memory as opposed to traversing the whole ship again after, which helps reduce the run time of my algorithm.

To visualize what bot3 is actually doing, we can take a look at he following heatmaps (images were found online) where the **bot is centered in the square**.
If a beep is heard, the probability updates will give larger weight to cells near



the center of the square grid. This is depicted on the **left**. If a beep is not heard, the probability updates will give larger weight to cells around and near the borders of the ship as shown on the **right**. While it is not guaranteed for theleak to



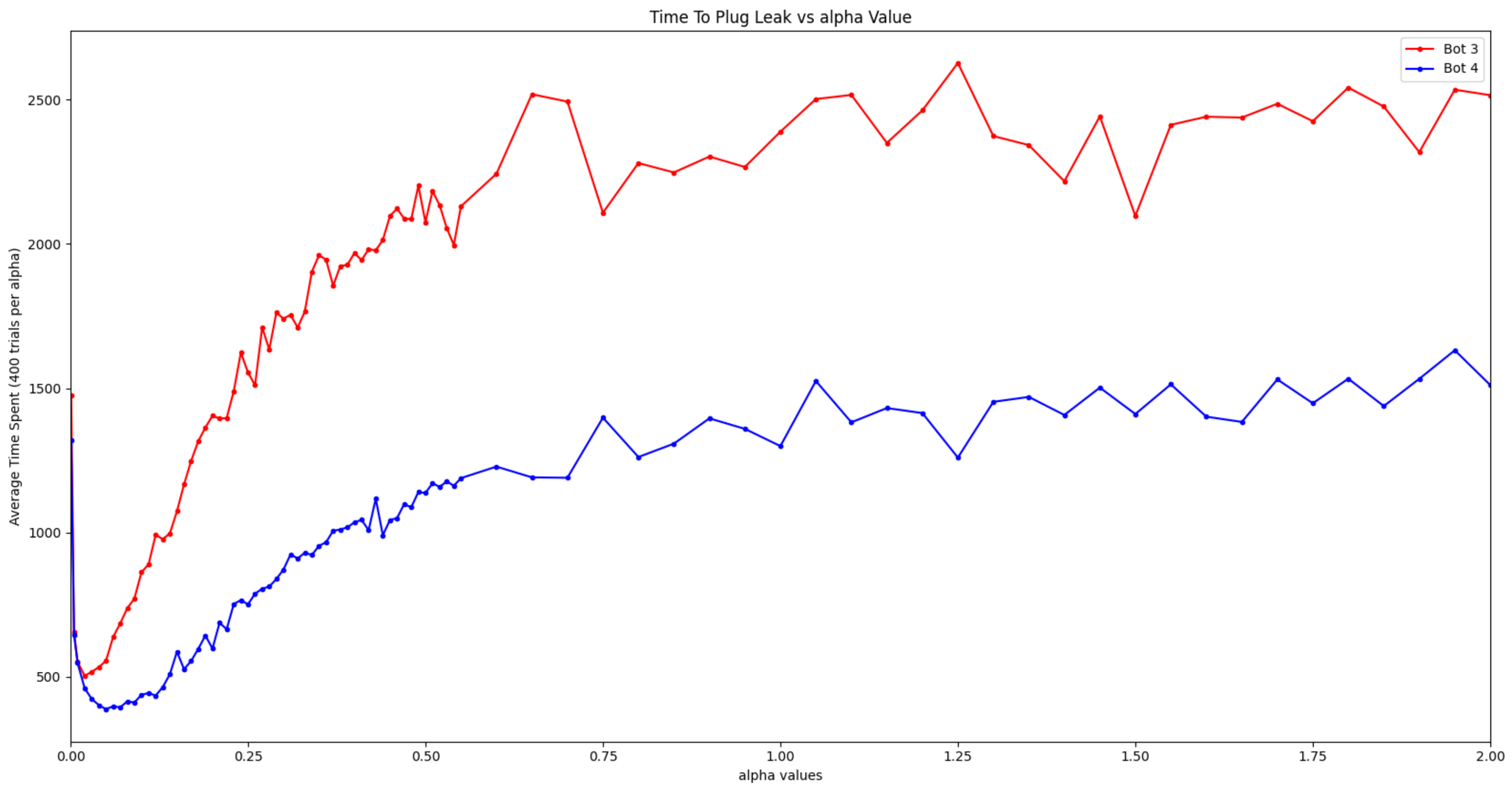
be in the red regions, Bot3 gives it self a higher probability of success in finding the leak if it travels to the *red* regions first.

Bot 4 -

This operates and has the same implementation algorithm in the test function as bot3, except for how we choose the next cell to travel to. All of the probability updates that are made are the same as derived in bot3 as well. The next cell in bot4 is chosen with the idea of local search in mind. Instead of choosing the cell with the highest probability in the *entire ship*, bot4 chooses the cell with the highest probability found only with the immediate 11x11 square grid with bot4 at its center. Since I implement my experiment with a 50x50 ship, this grid is a good size in searching for the next state.

The idea is hear is to minimize the chance of having the bot fully travel in the wrong direction. In bot3 if the maximum probability is on the other size of the ship, bot3 will travel there and check that cell. If it happens to be wrong, bot3 has wasted a full trip across the ship. Bot4 will start heading in that direction and scan again. If it’s knowledge continues to be reinforced with beeps or no-beeps, the bot will continue in that direction. Even though bot4 will scan more often, it save *huge* amounts of time, simply by *incrementally* moving towards cells of higher probability.

Shown is a graph comparing the performance of Bot3 and Bot4 (50x50 ship | 400 trials per α):



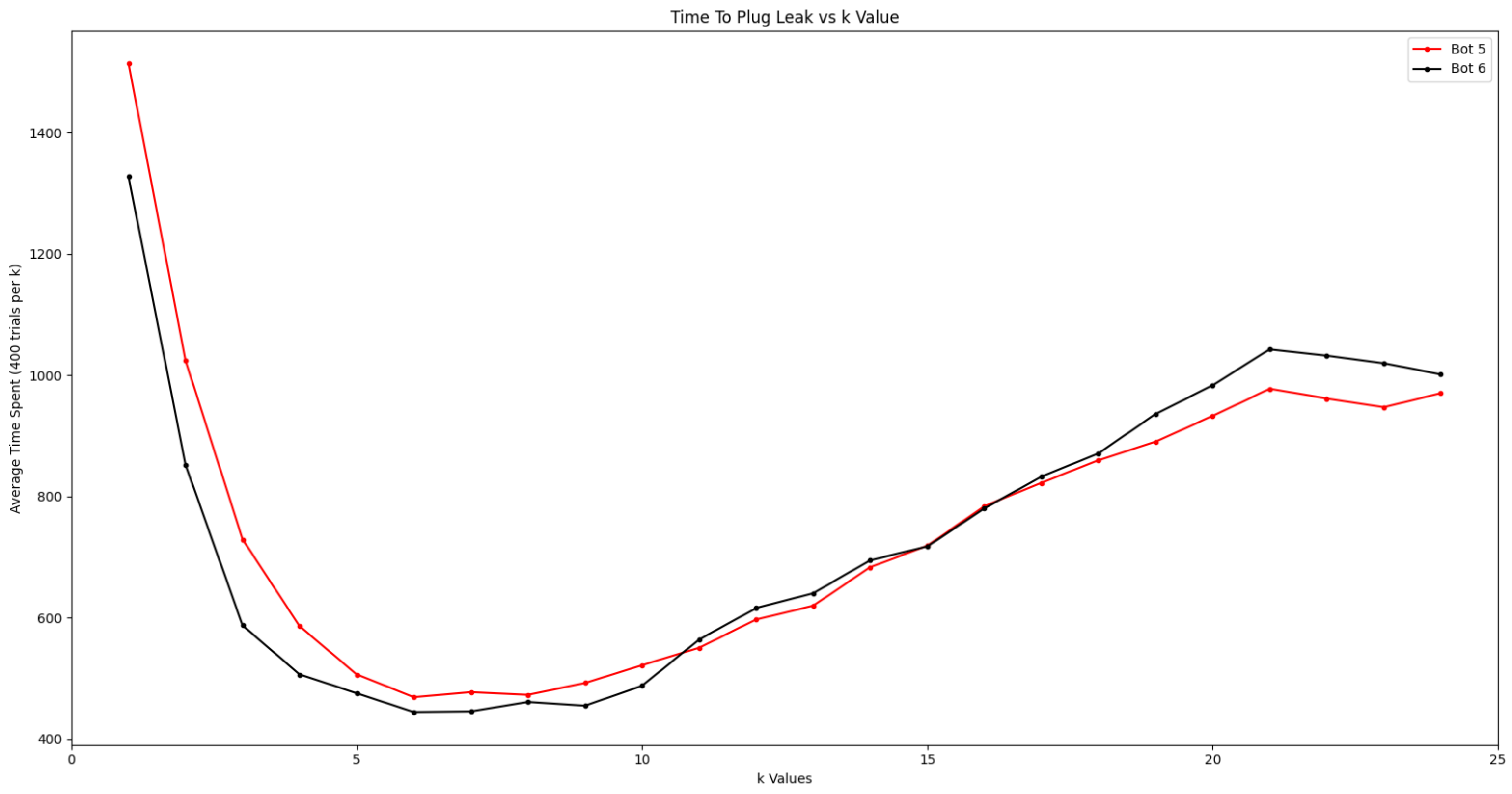
For α that are very close to 0, the performance is quite high for both Bot3 and Bo4. As α increases the performance of the bot quickly dips and then rises until the average time to plug the leak plateaus at around 2250 for bot3 and 1500 for bot4. Simply by incrementally moving towards our goal (local search) the average time spent to plug the leak is 30% better for bot4 than bot3 in the worst case (α > 0.7). Bot4 is even twice as fast as bot3 for quite a few α values.

Bot5 -
This bot is identical to bot1 in that it searches for one leak, finds that leak, and then proceeds to find the second leak using an implementation identical to bot1.

The only big adjustment that we have to make is due to the nature of data that is passed to the bot via the scanner. The scanner only give the bot a binary *yes detected* or *no detected*. If a leak is detected it does not tell bot5 *how many* leaks were detected. Therefore once a leak is detected the first time around, bot5 must visited very cell where a leak was detected to make sure that it does not miss a leak.

Bot 6 -
Similarly for bot6, the implementation is the same as Bot2. Since we learned of an inconvenience in Bot5 where we have to search every cell in the detected grid, for Bot6 I changed the implementation so that bot6 terminates it's search once it has located one leak instead of continuing to search the grid where a leak was detected. The bot then scans once more to check if the second leak is in the area. This saves the time cost of continuing the search by traveling to every cell, and instead lets the bot know if the second leak is near it by using only one scan time-step.

Below is a graph comparing the performance of Bot5 and Bot6 (50x50 ship | 400 trials):



Similar to bots 1 and 2, the performance of bot6 is much better than the performance of bot5 at least up until k = 10. From there the performance of the two bots is roughly the same, which bot6 deviating slightly away from the trend towards the end. After testing bots for both 1 leak and 2 leaks, it looks as though a k value between 6 and 8 are the optimal sizes for bot scanners. To summarize, both bot5 and bot6 utilize an A* algorithm for path planning to the next cell, while bot6 uses a local search approach towards finding the *best* and closest next state.

Bot 7 -
This bot is the first of the three double leak probabilistic bots. For *all* of the three bots (Bot7, 8, and 9) my algorithm has two parts. The first part is in effect only until the first leak is found. After the first leak is found, all three bots move into the second part. In the first part of the algorithm for these bots, updating probabilities is different than Bots 3 and 4. While after the first leak is found, the bots default to the standard way of updating probabilities as shows in Bot3. For bot7 we do not implement that change in updating probabilities. Bot7 treats the experiment as if there is only 1 leak, but only terminates when both leaks are found. It's algorithm is identical to Bot3

Bot 8 -
Bot 8 is exactly the same algorithm as Bot3 (where we choose the next cell based on the maximum probability of having the leak and use A* to get a path to that cell). However, in bot 8 we make the change in updating probabilities to accommodate for two leaks. When there is only 1 leak, bot 8 updates probabilities exactly like bot 3 and 4.

In order to update probabilities for the two leak situation, we can think about the problem differently. Let's have a probability that the leaks are cell J and cell K or in other words $P(Leak\ in\ J\ \cap\ Leak\ in\ K)\ =\ P(Leak\ in\ J\ and\ K)$. If we keep track of this for *every unique* pair then we can update these probabilities correctly. This will be the values that bot7 keeps track of. In my implementation I use a dictionary data structure (map) for this. For example: lets say J is (0,1) and K is (3,5) with a probability of 0.5. My data structure will hold a key of [(0,1),(3,5)] which points to a value of 0.5. Also important to note is that [(0,1),(3,4)] and [(3,4),(0,1)] should *not* both be considered because the leaks are identical and [(0,0),(0,0)] is invalid because both leaks cannot be in the same location.

So how do we get a list of unique pairs? Lets says we have an array: x = [0,1,2,3,4,5]
To get all unique pairs of number we can run the following code:

```
x = [0,1,2,3,4,5]
l = len(x)
for i in range(l):
   for j in range(i+1,l):
       print(x[i],x[j])
```

Since I keep record of all the possible cells that have not yet been searched as a list of tuples, we can generate a unique list of cell pairs in our ship.

Same as bot3, the probability of every pair having both leaks is equal at the start of the experiment. Therefore for every pair J,K the probability of both having a leak is given as

$$\frac{1}{Number\ of\ unique\ Pairs}$$

To update probabilities during the *double leak phase* of the experiment there are four cases to consider (Let A be the bot's current location)

1. Update when no leak found in A
2. Update when leak found in A
3. Update when *beep heard* while in A
4. Update when *no beep heard* while in A

We also need some equations of conditional probability:

- $P(X \mid Y) = \dfrac{P(X \cap Y)}{P(Y)}$
- $P(X \cap Y) = P(X) \cdot P(Y \mid X)$

Case 1 -

Making use of conditional probability all we need to do is update the probability for each pair J and K given that we didnt find a leak in A.

$$P(Leak\ in\ J\ and\ K \mid No\ leak\ in\ A)$$

$$= \frac{P(Leak\ in\ J\ and\ K \cap No\ leak\ in\ A)}{P(No\ leak\ in\ A)}$$

$$= \frac{P(Leak\ in\ J\ and\ K) \cdot P(No\ Leak\ in\ A \mid Leak\ in\ J\ and\ K)}{P(No\ Leak\ in\ A)}$$

If the leak is in J and K, it cannot possibly be in cell A, so $P(No\ Leak\ in\ A \mid Leak\ in\ J\ and\ K)$ must be 1

$$= \frac{P(Leak\ in\ J\ and\ K)}{1 - P(Leak\ in\ A)}$$

$$= \frac{P(Leak\ in\ J\ and\ K)}{1 - \sum_{all\ pairs\ (L,A)} P(Leak\ in\ L\ and\ A)}$$

$P(Leak\ in\ J\ and\ K)$ is the value in the map for (cell J, cell K) and we can find the sum of the probabilities for every pair that has cell A by looping over the dictionary object. In my implementation, after looping through every pair and finding the sum of the probablitie for every pair that has cell A, I remove these pairs from this map so that we do not have situation where we are counting probabilities for a cell where the bot is currently in.

Case 2 -
In this step my bot doesn't actually update any probabilities. Instead since we have found the first of the two leaks, in this step I make sure that the data structures and probabilities that I use to update probabilities like in bot 3 and 4 are genearted accurately in order to reset the problem and continue for the one leak situation.

Case 3 -
Using conditional probability we can update each probability for pair (J,K) by solving:

$$P(Leak\ in\ J\ and\ K \mid Beep\ in\ A)$$
$$= \frac{P(Leak\ in\ J\ and\ K \cap Beep\ in\ A)}{P(Beep\ in\ A)}$$
$$= \frac{P(Leak\ in\ J\ and\ K) \cdot P(Beep\ in\ A \mid Leak\ in\ J\ and\ K)}{P(Beep\ in\ A)}$$
$$= \frac{P(Leak\ in\ J\ and\ K) \cdot (1 - P(No\ beep\ in\ A \mid Leak\ in\ J\ and\ K))}{P(Beep\ in\ A)}$$
$$= \frac{P(Leak\ in\ J\ and\ K) \cdot [1 - (1-e^{-\alpha(d1-1)})(1-e^{-\alpha(d2-1)})]}{P(Beep\ in\ A)},$$

Where d1=len(A→J), d2=len(A→K)

$$= \frac{P(Leak\ in\ J\ and\ K) \cdot [1 - (1-e^{-\alpha(d1-1)})(1-e^{-\alpha(d2-1)})]}{\sum_{all\ pairs\ (M,N)} P(Leak\ in\ M\ and\ N) \cdot [1 - (1-e^{-\alpha(d3-1)})(1-e^{-\alpha(d4-1)})]}$$

Where d3=len(A→M), d4=len(A→N)

Note: $P(Beep\ in\ A \mid Leak\ in\ J\ and\ K)$ = the probability that we got a beep from leak1 + probability of beep from leak2 + probability of leak from leak1 and leak2. This quite lengthy. So instead we can calculate $P(Beep\ in\ A \mid Leak\ in\ J\ and\ K)$ as 1 - the probability that neither leak gave a beep.

Case 4 -
For this case the probability calculations are exactly the same as case 3 *except* we use the probability that A *does not* get a beep from either leak.

$$\frac{P(Leak\ in\ J\ and\ K) \cdot (1-e^{-\alpha(d1-1)})(1-e^{-\alpha(d2-1)})}{\sum_{all\ pairs\ (M,N)} P(Leak\ in\ M\ and\ N) \cdot (1-e^{-\alpha(d3-1)})(1-e^{-\alpha(d4-1)})}$$

Where d1=len(A→M), d2=len(A→N), d3=len(A→M), d4=len(A→N)

Now that we can update probabilities correctly, how does Bot 8 choose the next cell? It depends on how many leaks have been found. If 1 leak has been found, Bot 8 chooses the next cell by checking the probability in each cell in the grid and traveling to the cell with the maximum value. When no leaks have been found yet, Bot 8 instead chooses the *pair* with the maximum probability. Now thinking about this intuitively, we can only travel to a cell not a pair. But since we have selected a pair, the probability of each having a leak in that pair should be *equal.* Then we can just choose the closer of the two cells in that *max pair*.

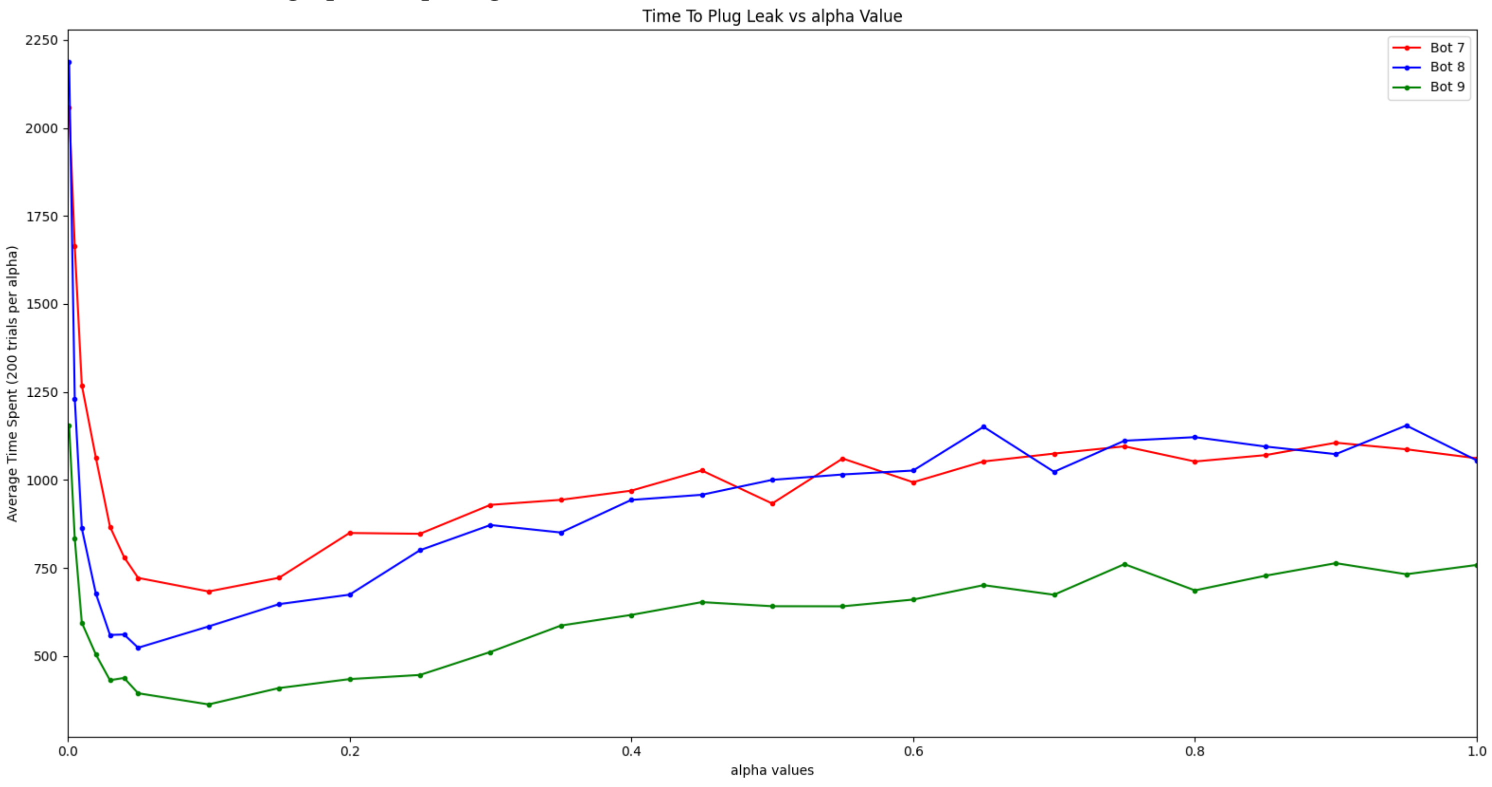
For bot's 8 and 9, I chose to reduce the size of my ship to 30x30 and run 200 trials per alpha value. Additionally I tested for only alpha value up to 1.0, since performance starts to plateau at around $\alpha = 0.7$. I made this decision in order to complete my simulations in a reasonable time. Previously for bot's 3,4 and 7, the big O runtime complexity is *O(n^2 * number of moves)* where n is the dimension value of the ship. However for Bot 8 and 9, since we have to keep track of every pair, the big *O(n^4 * number of moves)*. This can balloon to be a very large amount of time. A 30x30 ship provides solid results for bot's 8 and 9. For consistency, I also tested bot 7 on the 30x30 ship with 400 trials per alpha value.

Bot 9 -
For this bot, I implement all the probability updates the same way. The only difference between bot 8 and bo 9 is that for this bot I choose the cell with the highest probability within an 11x11 grid where the bot is centered in that grid.

When there are 2 leaks, the *pair* with the highest probability is chosen. Then bot 9 takes the cell in that pair that is closest. However, the catch is that at least on the cells in that pair must be within the bots immediate 11x11 grid surroundings. If there is no pair that exists, then bot 9 defaults to choosing the *max pair* from anywhere in the ship. When there is 1 leak, bot 9 chooses the next cell *exactly* like Bot 4.

The goal of this is to implement an idea of local search. In doing this, we do not need to waste valuable time by traveling very far if we end up choosing the wrong cell on the other side of the ship. Instead we incrementally move the bot and give it more of a chance to scan and access it's surroundings. Like bot3 and bot4, my bot scans as soon as it enters the goal cell and makes sure the leak is not there.

Below is a graph comparing bot 7, 8, and 9:



We can see that simply updating the probabilities in the correct manner for the two leak situation, automatically reduces the average time to success. For alpha values greater than 0.5, bot 7 and 8 are roughly the same. But using a local search approach like in bot 9 reduces average runtime of the bot significantly for all alpha values from 0.001 to 1.

Choosing k, α Values

k value -

For the probabilistic bot's, the size of my ship is 50x50. If k is greater than 24 then the range of the sensor could potentially span the entire ship. Since it is required that leaks be placed *outside* the initial sensor range, k values must be less than 24. k can be 0, but the bot would *highly* inefficient. So the ranke of k values I have chosen for all probabilistic bots is k ∈ [1, 24].

α value -

The alpha value basically determines how sensitive our sensor is. If alpha is very close to 0, the bot will be able to detect leaks that are very far a way and have high probabilities of getting a beep for far cells. When alpha is larger ($\alpha > 1.0$) the the leak almost has to be next to the current cell or withing 2 cells in order for the bot to receive a beep. In both extreme cases, it is not good for the bot and performance suffers. It's the reason that in all deterministic bots the performance starts very poorly because the α that is tested is 0.001. It's also the reason that for $\alpha > 0.8$ the performance is relatively poor compared to the optimal α value of 0.1. Performance also plateaus when α is greater than 1.0. For bots 3 and 4 i tested all the way to $\alpha = 2.0$ to show that there is plateau. For bot's 7, 8, and 9 I only test till 1.0 in the interest of time. To summarize the α value that I test for bot 3 and 4 are

```
α ∈ [0.001, 0.005, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1,
0.11, 0.12, 0.13, 0.14, 0.15, 0.16,
        0.17, 0.18, 0.19, 0.2, 0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28,
0.29, 0.3, 0.31, 0.32, 0.33, 0.34, 0.35,
        0.36, 0.37, 0.38, 0.39, 0.4, 0.41, 0.42, 0.43, 0.44,0.45, 0.46, 0.47,
0.48, 0.49, 0.5, 0.51, 0.52, 0.53, 0.54,
        0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0, 1.05, 1.1, 1.15,
1.2, 1.25, 1.3, 1.35, 1.4, 1.45, 1.5,
        1.55, 1.6, 1.65, 1.7, 1.75, 1.8, 1.85, 1.9, 1.95, 2.0]
```

For bot's 7, 8, and 9 the α values are

```
α ∈ [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6,
           0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]
```

The Ideal Bot -

My idea of an ideal bot would be one that could process data and update probabilities much faster. One of the biggest problems I had was keeping the runtime low in my implementations. There were optimizations that I was able to make, such as calculating the maximum probability cell *while* updating probabilities to save an extra grid transversal. However my ideal bot would compute probabilities and update probabilities in real time. For probabilistic bots my bot would scan rows in parallel to save time in scanning larger ships.

If we can optimize the ideal bot for speed via parallel computing and distributed computing, we can afford to scan the board in a single spot multiple times. The more we scan the better "heat map" we can generate of probabilities. If we scan in a single location say 5 or 10 times, it will not cost us as much as moving to the completely wrong side of the ship. In running this experiment, I found that the most costly parts to the bot's average time to success was the number of moves that it made. Sometimes the bot would have to back track through cells it had already searched. If we can maximize computer power, we can form a better picture for the bot.