Intro to AI

Joshua Redona, David Harianto

November 15, 2023

Bot 1 vs Bot 2

Bot 1

For bot1, we first initialize the graph and randomly assign an open cell to be where the bot is. Then using a function called placeLeak(), we place the leak outside of the bot's detection zone. If the bot's detection zone is too big (k is large), then we randomly assign any open cell to be where the leak is. We will initialize an array called MAY CONTAIN LEAK that will contain all the open cells and represents the nodes we haven't looked at yet. After that, we will then scan for the leak in our detection radius using check detection radius() before entering a while loop that will only break when we find the leak.

If there is a leak, we will mark the leak as found, otherwise, we will then proceed to eliminate all the cells in our detection radius from our MAY CONTAIN LEAK array. This method uses two for loops that go through (x–k) to (x + k) values on the x axis and (y–k) to (y + k) values on the y axis. This will create the detection square that will look through all nodes that are in MAY CONTAIN LEAK. We will add any nodes we find in MAY CONTAIN LEAK and return that list of nodes and return whether we found the leak or not. We will add +1 action for this scanning action.

If the leak is found, then we will proceed to search only through the unchecked nodes in our search radius using find closest node(). This method will return the closest unchecked node and will return the distance of that node. The bot will go to that node, and we will add that distance to actions as the number of spaces the bot moved. If the bot lands on the leak, we will break out of the loop and return the number of actions. Otherwise, we will remove the node we traveled to from our search radius. We will continue searching through all the unchecked nodes in our search radius until we find the leak. On the other hand, if we didn't find a leak from our scan earlier, then we will go to the closest node in MAY CONTAIN LEAK using find closest node() like mentioned earlier.

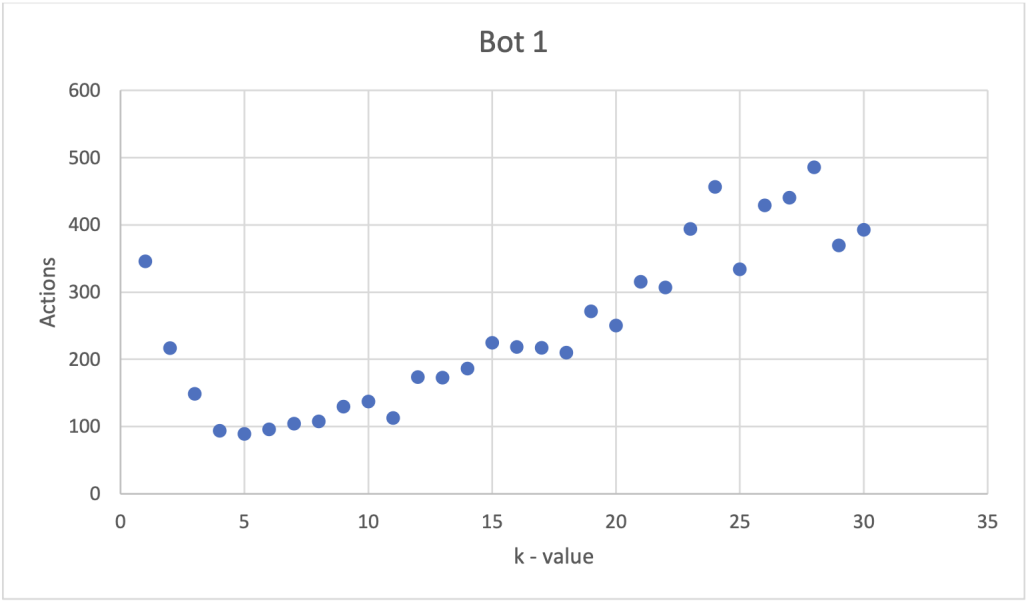
Since we eliminated the search radius from our MAY CONTAIN LEAK earlier since we didn't find a leak, this method will land our bot to the closest unchecked node outside our radius. We will get the closest node and the distance, and we will update our actions count. If we find a leak, we break out of the loop and return the number of actions. If we didn't step on a leak, we remove our current cell we are on from our MAY CONTAIN LEAK and scan. If we scan no leaks, remove the new detection radius cells from our array MAY CONTAIN LEAK and go through the loop again until we find the leak. For our find closest node() method I mentioned earlier, it uses a priority queue to keep track of the nodes and their distances. We also have variables closest nodes and closest distance to keep track of.

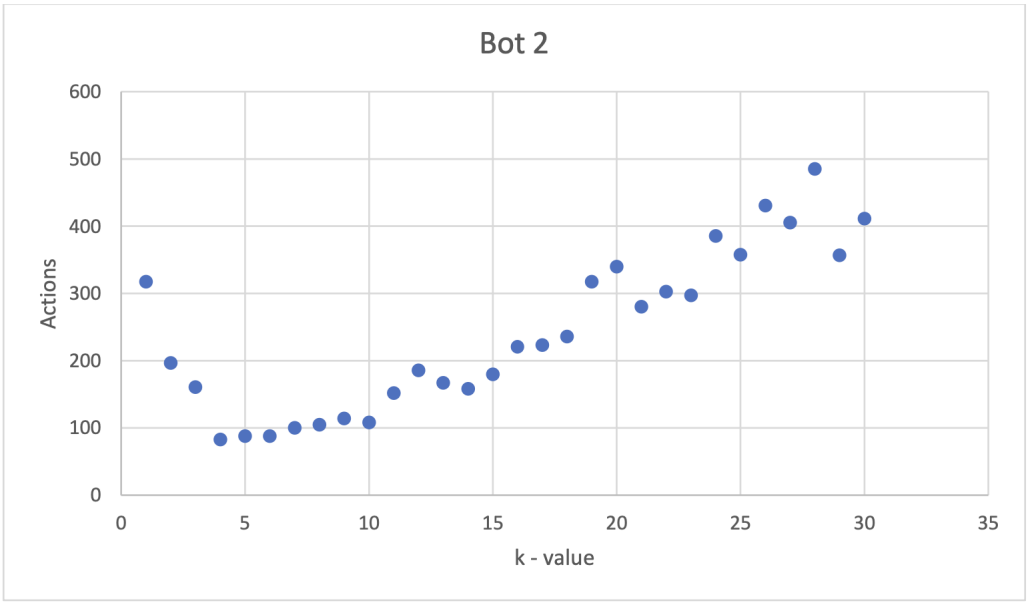
Using BFS, we will pop the node with the smallest distance first. If it has been visited already, we will skip, otherwise, we will add that node as visited. If the node is in the array that we provided in the parameter (it will either be MAY CONTAIN LEAK or detection range), check if the distance is less than or equal to the current closest distance. If it is less than the closest distance, we will clear the closest nodes array and add this node as a new entry. If it is the same, we will add it to the array. Then we will append all the neighbors of the current node to the queue if they haven't been visited yet. Once the queue finishes, we will choose a random node from our closest nodes array and return that node and its distance.

Bot 2

For bot2, it will follow the same exact formula as well. The only difference is that when we don't find a leak, instead of using find closest node(), we will use the method called find closest best node(). find closest best node() follows almost the same exact code as find closest node(), but the main difference is that instead of returning a random node from our array of closest nodes, we will instead choose a node that has the most unchecked cells around it. Using a method called count unchecked nodes in radius(), we will count the amount of unchecked nodes around a given node, and return the node with the most unchecked cells around it (and its distance). Bot2 improves upon bot1 by reducing its randomness when choosing the next node when there are many nodes with the same distance for when a leak hasn't been detected yet. This only applies though when the leak hasn't been found. If the leak has been found, we will simply use the same method in both bot1 and both2 which is find closest node().

Comparisons





In our graphs for bot1 and bot2, we can clearly see that both follow a distinct U-shaped pattern. Actions for both start off high at the lowest k-values, then actions decrease as k-values start to increase, but at a certain point, actions start to increase rapidly as k-values become bigger and bigger. We can subtly see that bot2 is a bit better than bot1. By reducing a bit of the randomness of bot1, our bot2 does help decrease the number of actions for bot1. The reason why bot1 and bot2 start off with high actions is because the search radius is much too small to be of any use. As k-values start to increase a bit, our search radius starts to become bigger and is much more useful at narrowing down the cells to find the leak. But much bigger k-values cause our bots to have more actions due to the search radius

being much too big. With the search radius being too big, now if we find the leak, it's much harder to eliminate the spaces since the leak is in this huge radius.

Bot 3 vs Bot 4

Bot 3

Bot 3 is the first of the probabilistic bots that we worked on. As such, we required a few more methods to make testing less time consuming.

First, we introduce a dictionary of all distances between cells i and j, and have named the dictionary 'allDist' in our methods. It is a simple look up table that we create when starting any of the probabilistic bots, and we initialize it by calling Bot3.dijkstra(), which runs dijkstra's algorithm from every square and returns the full distance map for each square. In comparison to running BFS search at every step, this is a much quicker way to get distances.

We have a beep function for the bot to run, which follows the formula given in the writeup. We exactly use the probability

```
p = math.exp(-a*(d-1))
```

and utilize this same probability for our update functions. We keep track of a probability map (usually referred to as the heatMap in the code) and have 3 updating functions for it. We update this heat map whenever the bot enters the cell, when the bot gets a beep and when the bot does not get a beep.

When entering a cell, the only probabilities we need to update are the cells directly adjacent to the bot, since we are given that if the bot is directly next to the leak, we will always get a beep. Then if we have not found the leak next to this cell, we set all of the values around it to 0 and normalize the heat map.

When the bot recieves a beep, we update the probability function with the following lines.
Beep:

```
prob[x] = ( prob[x] * pBeep) / totalProb
```

No Beep:

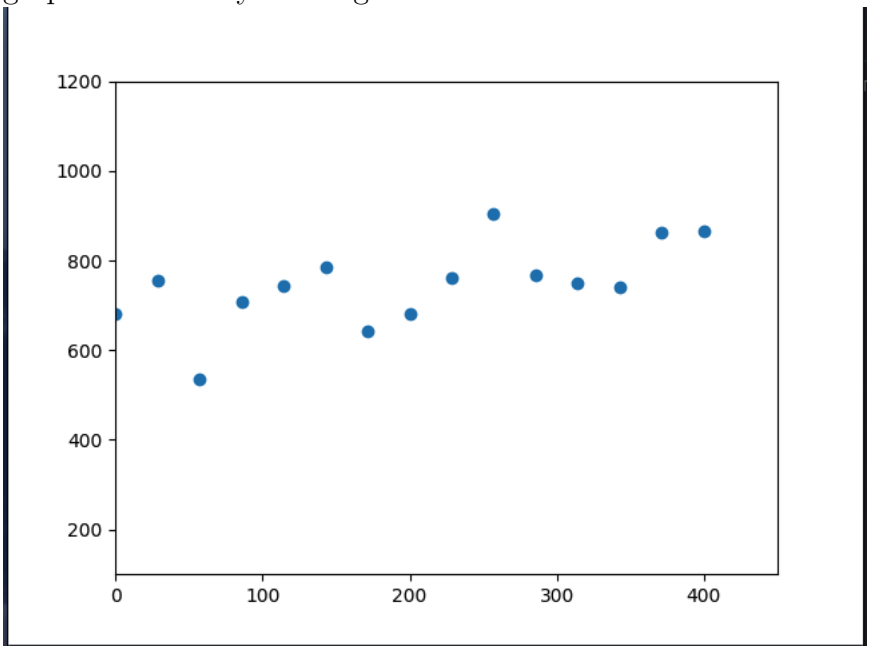
```
prob[x] = ( prob[x] * (1 - pBeep)) / totalProb
```

The probability calculations we used come from using Bayes Theorem,

$$P(x|y) = P(x) * P(y|x)/P(y)$$

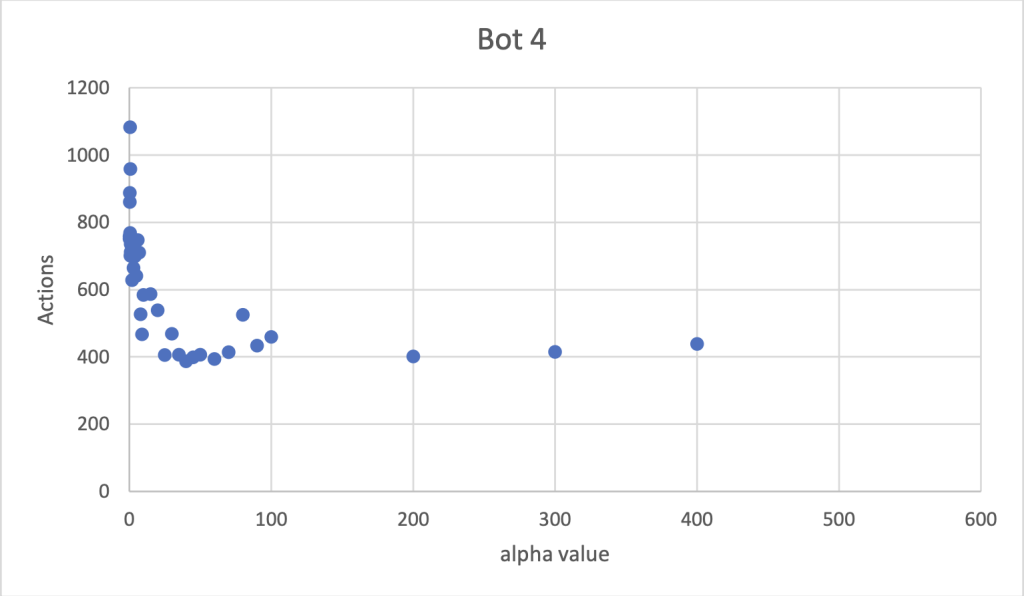
where x and y are the events recieving a beep and the leak being in cell i respectively.

At every step, the bot will look at the heatmap and get the highest probability it finds on it and plan a path to that cell. If at any point the highest probability cell changes, then the bot will replan a path and begin heading towards the new target. The bot stops once it is in the same cell as the leak. The following is the graph obtained by running Bot 3.



There are 15 evenly spaced points between alpha values 0 to 400. Each point is taken by averaging 10 trials of Bot 3 with that alpha value.

Bot 4



For bot4, we first initialize the graph and randomly assign an open cell to be where the bot is. Each bot will have an instance variable called succeeded which will let us know if we have found the leak or not. Then using a function called placeLeak(), we place the leak outside of the bot's detection zone. If the bot's detection zone is too big (k is large), then we randomly assign any open cell to be where the leak is. We will initialize an array called MAY CONTAIN LEAK that will contain all the open cells and represents the nodes we haven't looked at yet. Then we will also initialize a graph of probabilities for every open cell. Using a method called initProbability, it will initially set all the open cells with probabilities of (1 / number of open cells) while the rest of the cells will be set to 0.

So while b4 has not found the leak yet, or b4.succeeded is not true, we will first begin our first action by checking for any beeps (actions +1) by calling our method beep(). This will first get the distance of the bot's current index to the index of the leak. This was calculated by finding the length of the shortest path and the shortest path was calculated by using BFS and using a dictionary to keep track of the parents of each node in the shortest path (calling getShortestPath() to get the array of the path indexes then calling getDistance() to find the length of the path). Depending on whether the bot hears the bot or not, we will change the probability graph accordingly. We will be using the same probability formula as Bot3 to replace all the old probabilities with new ones. If there is a beep,

prob[cell j] = [prob[cell j] * prob(beep in curr cell — leak in cell j)] / prob(beep in curr cell).

If there is no beep,

prob[cell j] = [prob[cell j] * prob(no beep in curr cell — leak in cell j)] / prob(no beep in curr cell). After the probabilities are updated, we then call the

method getHighestProbabilities() which just goes through the probabilitiy map and returns a list of the highest probabilities. From this list, we will call find closest best node to choose the best node to go to (this method is the same as the previous methods used in bots such as bot2). So from these nodes that have the highest probabilities and same distance, we will choose the node that has the most unchecked cells around it (most cells around the given node that have probability ¿ 0). Once we get that node, we will use getShortestPath() (BFS traversal) to get the array of shortest path of nodes between current bot cell and the best node. As the bot traverses one cell at a time (actions +1), change the probabilities accordingly. If the bot lands on the leak, quit the loop and return actions, otherwise, continue the loop again.

Comparisons

From the graphs alone, it is plain to see that Bot 4 performs significantly better for higher alpha values. Although, while Bot 4 initially starts off with poor performance in $\alpha < 100$, Bot 3 initially does better, averaging under 1000 actions per simulation. While Bot 3 seems to level off and take between 600-800 actions no matter what α we are working with, Bot 4 seems to perform better with higher values of α.

Bot 5 vs Bot 6

Bot 5

For bot5, we first initialize the graph and randomly assign an open cell to be where the bot is. For these double leak bots, we have another instance variable called succeeded2 which will let us know if we have found both leaks. Then using a function called placeLeak(), we place 2 leaks outside of the bot's detection zone. If the bot's detection zone is too big (k is large), then we randomly assign any open cell to be where the leak is. We will initialize an array called MAY CONTAIN LEAK that will contain all the open cells and represents the nodes we haven't looked at yet. After that, we will then scan for the leak in our detection radius using check detection radius() before entering a while loop that will only break when we find the second leak. If there is a leak, we will mark the leak as found, otherwise, we will then proceed to eliminate all the cells in our detection radius from our MAY CONTAIN LEAK array. This method uses two for loops that go through (x – k) to (x + k) values on the x-axis and (y – k) to (y + k) values on the y-axis. This will create the detection square that will look through all nodes that are in MAY CONTAIN LEAK. We will add any nodes we find in MAY CONTAIN LEAK and return that list of nodes and return whether we found the leak or not. We will add +1 action for this scanning action.

If the leak is detected, then we will proceed to search only through the unchecked nodes in our search radius using find closest node(). This method will return the closest unchecked node and will return the distance of that node. The bot will go to that node, and we will add that distance to actions as the number of spaces the bot moved. If the bot lands on the leak, we first check if this is the first leak found or the second. If it is the first leak, we will mark that down in the bot's succeeded variable and we mark leak found to be false. Then we will remove the current index from MAY CONTAIN LEAK and scan again to check if there might be 2 leaks in the same radius (this will add +1 to actions). If this is the second leak, we will break out of the loop and return the number of actions. If the bot doesn't land on a leak cell, we will remove the current index from MAY CONTAIN LEAK and the detection radius we are searching. We will continue searching through all the unchecked nodes in our search radius until we find the leak(s).

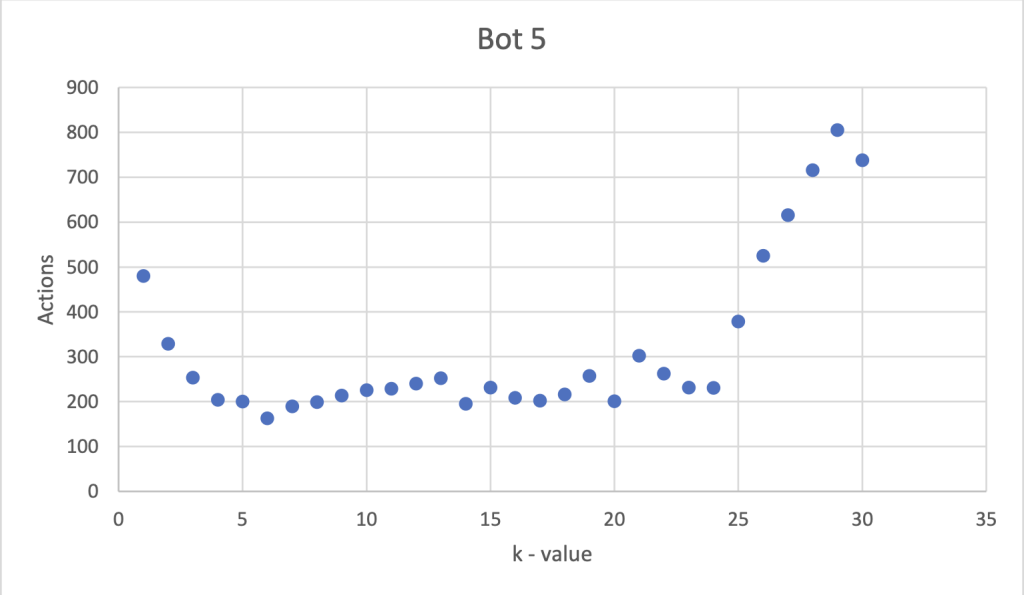
On the other hand, if we didn't find a leak from our scan earlier, then we will go to the closest node in MAY CONTAIN LEAK using find closest node() like mentioned earlier. Since we eliminated the search radius from our MAY CONTAIN LEAK earlier since we didn't find a leak, this method will land our bot to the closest unchecked node outside our radius. We will get the closest node and the distance, and we will update our actions count. If we find the second leak, we break out of the loop and return the number of actions. If we didn't step on a leak, we remove our current cell we are on from our MAY CONTAIN LEAK and scan (actions +1). If we scan no leaks, remove the new detection radius cells from our array MAY CONTAIN LEAK and go through the loop again until we find both leaks. For our find closest node() method we mentioned earlier, since bot5 is a subclass of bot1, it uses the same method. So, it has the same code and same function, so the method will choose a random node from our closest nodes array and return that node and its distance (using a priority queue and DFS).

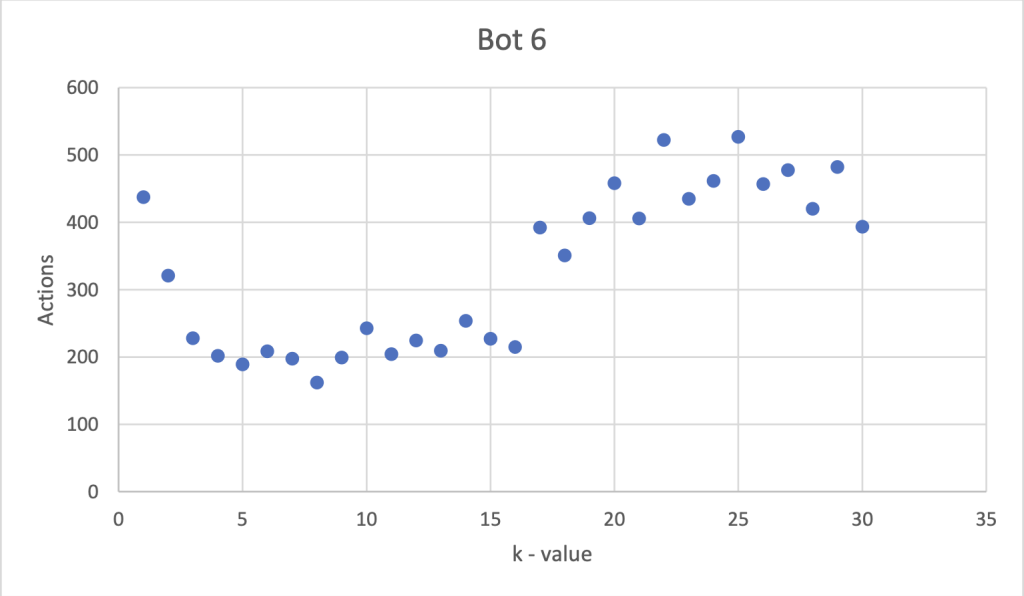
Bot 6

For bot6, it will follow the same exact formula as bot5. The only difference is that when we don't find a leak, instead of using find closest node(), we will use the method called find closest best node(). This method follows the same code as bot2's method. So like we mentioned earlier, this method will follow the same exact code as find closest node(), instead of returning a random node from our array of closest nodes, we will instead choose a node that has the most unchecked cells around it. Using a method called count unchecked nodes in radius(), we will count the amount of unchecked nodes around a given node, and return the node with the most unchecked cells around it (and its distance). Bot6 improves upon bot5 the same way bot2 improves upon bot1, by reducing its randomness when choosing the next node when there are many nodes with the same distance for when a leak hasn't been detected yet. This only applies though when the leak hasn't been found. If the leak has been found, we will simply use the same method

in both bot5 and bot6 which is find closest node().

Comparisons





In our graphs for bot5 and bot6, we can clearly see that both follow a distinct U-shaped pattern. Actions for both start off high at the lowest k-values, then actions decrease as k-values start to increase, but at a certain point, actions start to increase rapidly as k-values become bigger and bigger. We can subtly see that bot6 is a bit better than bot5. By reducing a bit of the randomness of bot5, our bot6 does help decrease the number of actions for bot5. The reason why bot5 and bot6 start off with high actions is because the search radius is much too small to be of any use. As k-values start to increase a bit, our search radius starts to become

bigger and is much more useful at narrowing down the cells to find the leak. But much bigger k-values cause our bots to have more actions due to the search radius being much too big. With the search radius being too big, now if we find the leak, it's much harder to eliminate the spaces since the leak is in this huge radius. For bot5 and bot6, I did 30 trials and got the average actions for every k-value from 1 to 30.

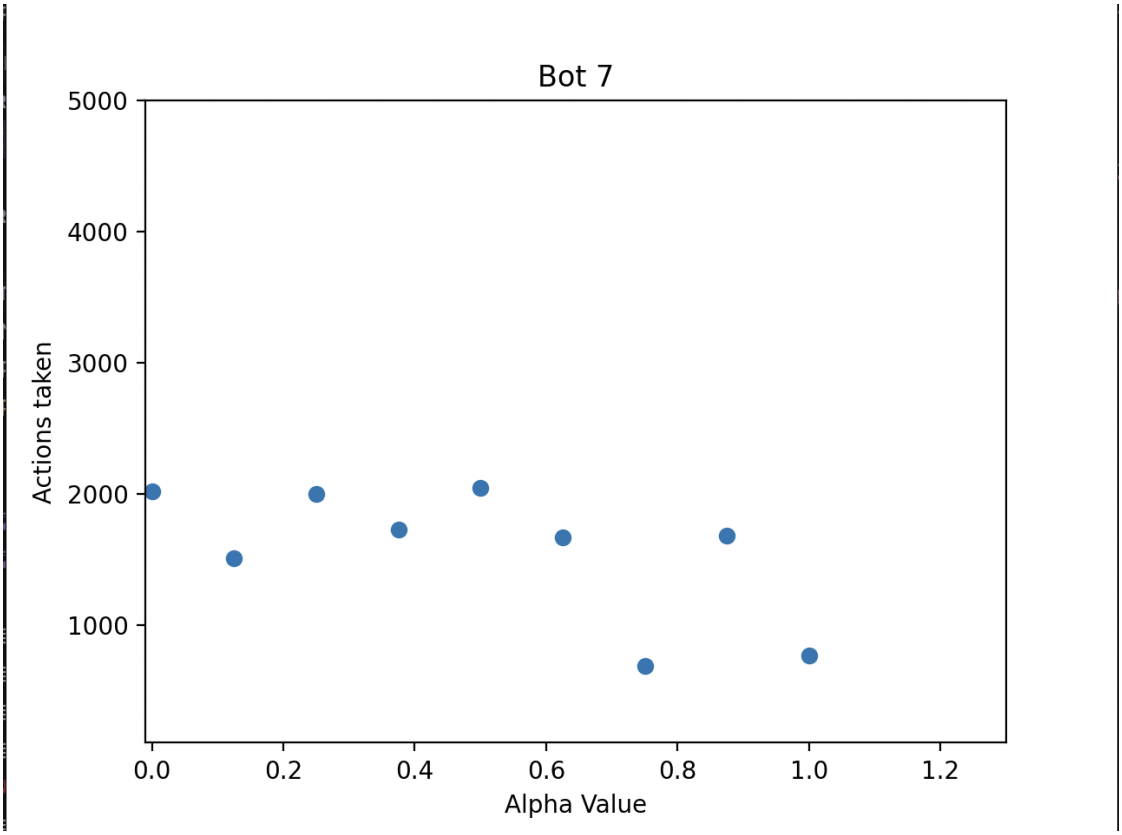
Bot 7 vs Bot 8 vs Bot 9

In an effort to keep the graphs consistent, each graph has the same axis scaling and the same number of data points. Alpha values range from 0 to 1. Each point on the graph corresponds to a single alpha value averaged over 10 simulations each. The size of each graph was 30x30, which was chosen due to time constraints. Each simulation was capped at 5000 actions, since a normal run normally takes much less to complete and outliers would skew the data harshly.

Despite these constraints, we still observed a large variance in our results, which will be visualized through the following graphs. It is worth noting that at the time, we had assumed that alpha can only range from 0 to 1, and so our data will only span these alpha ranges for bots 7, 8 and 9.

For each of the following, we initialize the graphs in the same way we did with the previous bots. 2 leaks are chosen at random, following the same constraints as the previous bots, and we treat them as ordered leaks, meaning for example, in bot 7, we choose the leak we initialize first as the first target, regardless of which one is closer to the starting location of the bot. This is less important for Bots 8 and 9 since our goal with 2 leaks is to find 1 leak and then set the next leak as the next target.

Bot 7

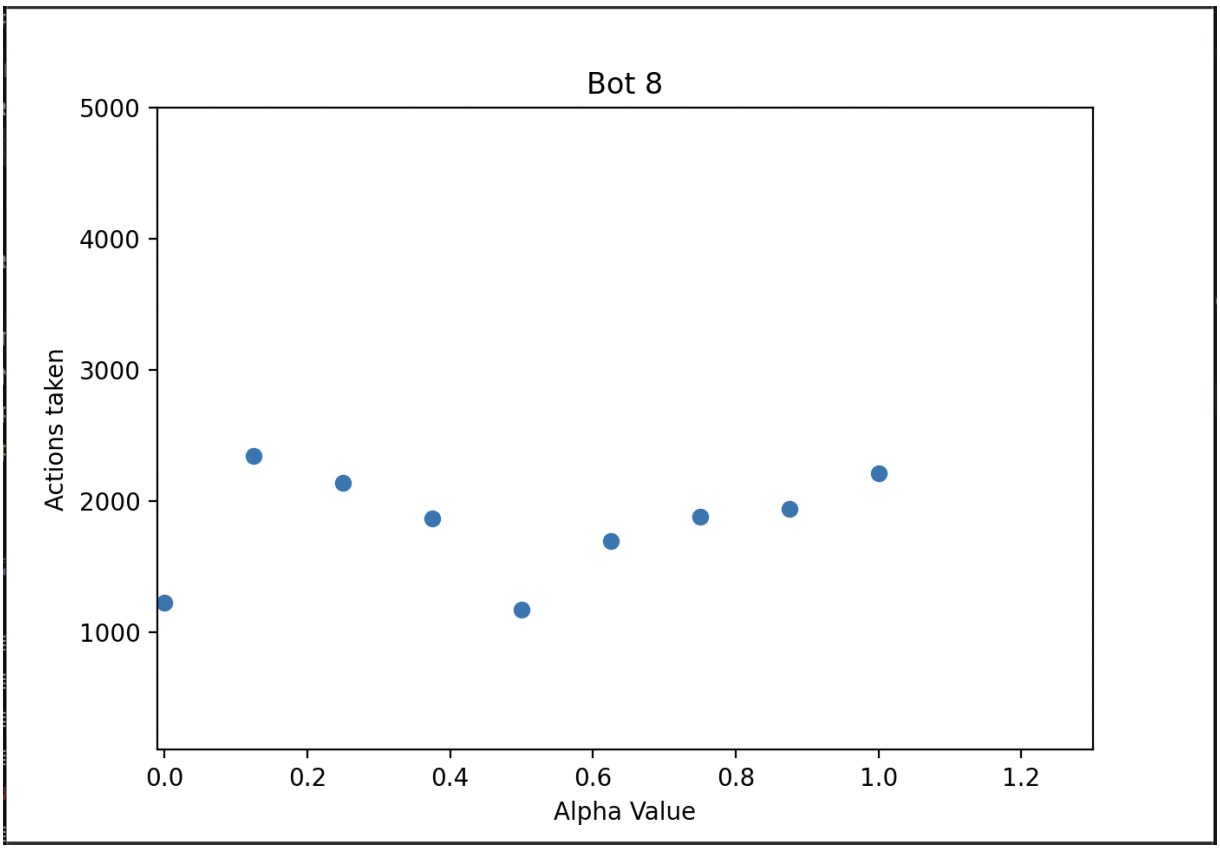


From the graph, we can see that Bot 7 ranges from around 680 actions to 2000 actions taken during a run. Bot 7 inherits all of Bot 3's methods and utilizes the exact same loop to find each leak.

Since there are two leaks to plug, two probability maps are initialized. The bot treats the two leaks as two seperate cases of Bot 3. We order the leaks on initialization, use Bot 3's loop to find the first leak and then completely disregard the updated probability map that we created. After one leak has been found, the second leak is used for probability calculations and runs exactly as Bot 3 would run for the one leak case.

Note that since Bot 8 handles the correct probability updates for two leaks, we chose to write the easiest possible bot that can handle 2 leaks in a probabilistic manner.

Bot 8



Values of the graph range from around 1200 actions to 2200 actions. It is considerably more actions than we see on Bot 7, and we can only speculate as to why.

Per the announcement, we used the formula to update our probability table as followws:

$$P(x|y \wedge z) = (1 - P(x|y))(1 - P(x|z))$$

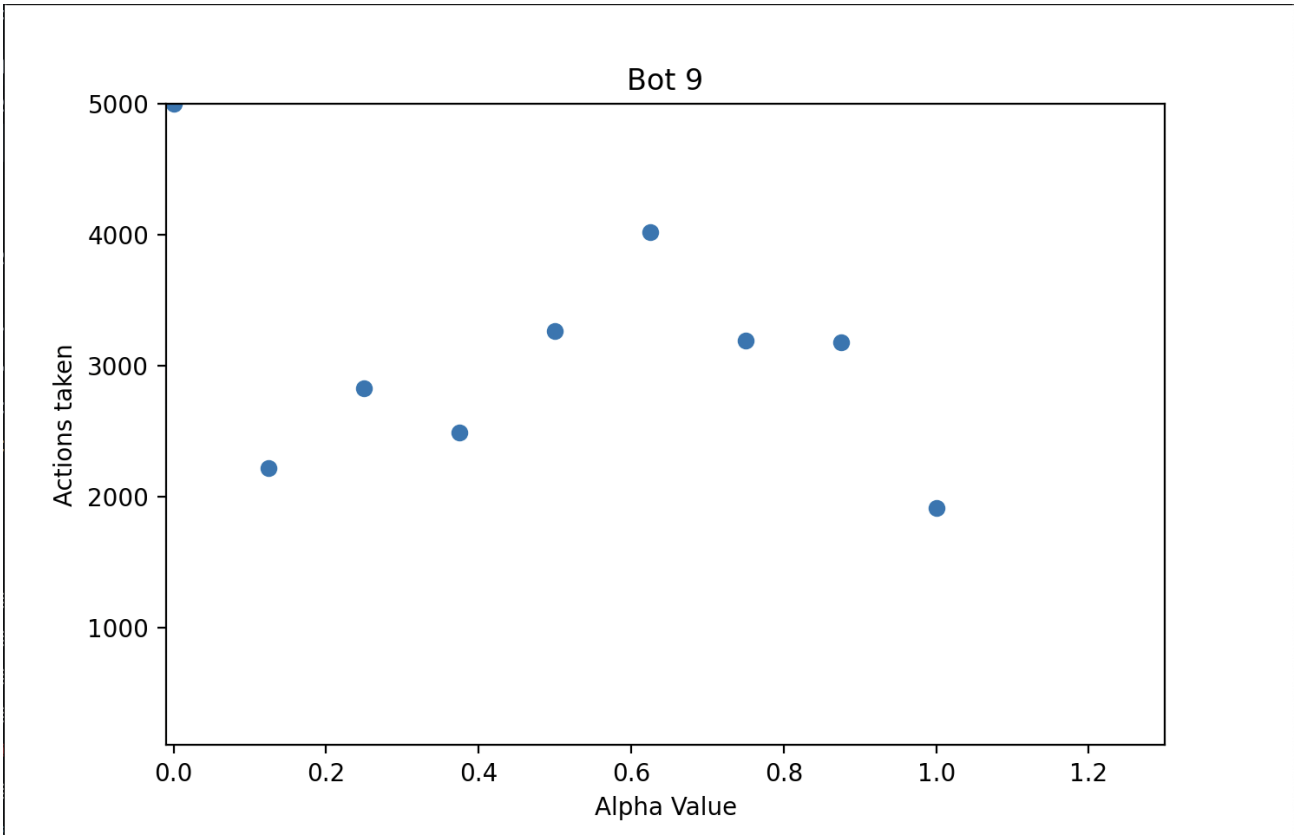
where x,y and z are the events beeping, a leak in j and a leak in k respectively. Admitedly, based on the inefficiency of the bot, it is possible that our coded updated formula does not reflect the on-paper probability update formula, which could have resulted in more actions needed to be taken to find a leak. For reference, the exact expresion used is

```
totalProb += prob[x] * (1 - math.exp(-a*(dx-1))) *
    (1-math.exp(-a*(dy-1)))
prob[x] = ( prob[x] * pBeep ) / totalProb
```

where dx and dy are the calculated distances to leak j and k. Note that some of the calculations here are to normalize the new probability that we calculate. Regardless, Bot 8 uses this 2 leak updating formula to keep track of one probability map. When it finds one of the leaks the bot initializes a new probability map and proceeds to find the next leak using Bot 3's method for 1 leak.

At first, we attempted to correct the probability map for 2 leaks to account for 1 leak missing, however this led to repeated loops around the first leak, and the idea was scrapped to save time. Ideally, finding a way to correct the heatmap would be the ideal solution to the 1 leak portion of Bot 8.

Bot 9



Our results for Bot 9 were significantly worse than for Bot 8. The values of the graph range from 2000 to 5000, which was the hard limit we kept for the bots, although the methods used by bot 9 may explain such a poor observation.

Bot 9 utilizes Bot 8's probability updating method, however it takes each step to beep 5 times. We had hoped that the extra beeps and updates to the probability map would make the map more accurate, possibly improving the results of Bot 8 by having a better map. We failed to account for the possibility that our update function was flawed, which would be compounded by running a flawed function multiple times.

In addition, at $\alpha = 0$, Bot 9 would sit in one spot and never recieve a beep, which means minute changes in the probability map. Nevertheless, when beeps were account for for other alpha values, Bot 9 completed the task, albeit with

many more actions than the other bots.

It appears as though higher alpha values work better for Bot 9, and had we tested more values for alpha, Bot 9 may have stayed competitive with the other bots.

Ideal Bot

If we are specifiying for 2 leaks (because that is the harder problem), then a corrected Bot 8 would likely be closest to the ideal bot. Specifically, if we are able to properly update the probability map from 2 leaks to 1 leaks, we could make the bot find both leaks much quicker.

Additionally, an ideal bot would be able to tell when it should start sensing again. Rather than using an action at every step to sense, an ideal bot would only sense when certain conditions are met, say like when the probability map has a new highest probability and it is in a different direction than the current target. Sensing when many of the highest probability squares are in the same direction as the bots current path, then sensing is just a waste of actions.