Experiment Implementation

Overview:

A run of the experiment is captured in an instance of the ExperimentController class. It holds references to the ship, bot, and leaks which are randomly generated by an appropriate spawn method. ExperimentController has 2 runExperiment methods, one for a single leak and another for two leaks. These run methods create the main loop of the experiment and handle the interactions between the bot and ship. The run methods will loop until the bot has found the leak(s) then return the number of actions the bot took.

Each bot has its own class which all inherit from the Bot abstract class. The Bot class implements common methods such as bfs searching through the ship and leaves other methods to be implemented by the bots such as sense. ExperimentController holds a reference to the Bot class, thus it implements a setter method for each bot that creates an instance of a Bot subclass and has the generalized reference point to it. These setter methods also initialize some prerequisites for each bot. For example, the bot 3 setter creates an instance of the BotThree class and initializes the probabilities for each cell in the ship.

The ExperimentController runExperiment methods call the botAction method which every bot implements. The method represents a single action of the bot where the bot determines whether to move or sense and processes new information gathered.

Each bot subclass also implements a setNoLeakOnBot method which is called from ExperimentController if botPosition doesn't equal the leak position. **Note that functionality such as updating cell probabilities after setting a cell's probability to 0 or marking a cell as not containing the leak after traversing it is not in the botAction method but in the runExperiment methods.**

Bot Class

1. Bot properties
    a. An instance of the bot class contains the botPosition and botPath instance variables
    b. botPosition
        i. Holds a reference to a Tile object. A Tile represents one cell on the ship and its relevant properties. The bot "moves" by having botPosition point to a different tile on the ship
    c. botPath
        i. Refers to an arraylist of Tile objects. Represents the path the bot takes to a desired cell starting at its current position

ii. Often updated with the Bot class's bfs algorithms.

2. bfsNotInSet, bfsInSet

    a. The bot class implements these two methods for all the bot subclasses to use. They both take parameters ship, set, list, startingTile.

    - i. ship: A reference to the experiment's ship
    - ii. set: In bfsInSet, it represents the set of all goal cells. In bfsNotInSet, it represents the set of all cells that aren't the goal cells.
    - iii. list: Both algorithms write the path to the closest goal cell to an arraylist which this parameter is a reference to.
    - iv. startingTile: The tile where the algorithm starts.

    b. Implementation:

```
    i.    queue = new Queue
    ii.   visited = new Set
    iii.  parent = new HashMap <Tile, Tile>
    iv.   botNeighbors = new Set
    v.    currentTile = startingTile
    vi.   queue.add(startingTile)
    vii.  visited.add(startingTile)
    viii. parent.put(startingTile, null)
    ix.   while queue not empty
            1. curr = queue.poll
            2. if(curr is an open tile and set contains curr)
                        i. #The if condition changes to (curr is an open
                           tile and set does not contain curr) for
                           bfsNotInSet
                  b. currentTile = curr
                  c. break;
            3. fill botNeighbors with open neighbors of curr
            4. for each cell in botNeighbors
                        i. #This additionally handles the behavior of
                           breaking ties in distance randomly since a set
                           is iterated through at random.
                  b. if visited does not contain cell
                        i.   queue.add(cell)
                        ii.  visited.add(cell)
                        iii. parent.put(cell, curr)
            5. clear botNeighbors
    x.    while(currentTile is not null)
            1. list.add(currentTile)
            2. currentTile = parent.get(currentTile)
    xi.   reverse the list
    xii.  remove first element of list
```

    c. This algorithm exhausts the last added element in its fringe before processing the next. Thus, it always finds the shortest path to the goal node if one exists.

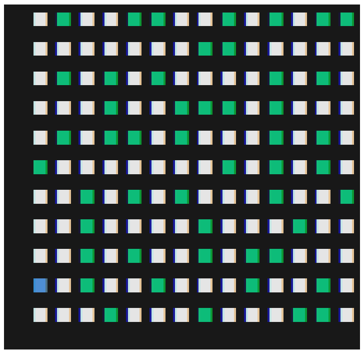
Bot 1

1. Overview
    a. Bot 1 uses a deterministic sense to find its way to the leak. It holds the set nonLeakTiles which gets filled with the cells that are ruled out of possibly containing the leak.
    b. The bot will repeatedly sense, then move to the nearest cell that is not a member of nonLeakTiles until the leak is found.
2. Sense
    a. Bot 1's initialization will take in a parameter k to determine the range of the sense.
    b. The sense method loops through all cells within a square with vertices (max(botX - k, 0)), (min(botX, shipEdgeLength)), (max(botY - k, 0)), (min(botY, shipEdgeLength))
        i. max and min methods prevent the sense from going over the edges of the ship
    c. If any of the cells are equal to the leak parameter, the method returns true.
3. botAction
    a. if (bot path is empty)
        i. if (sense) //sense returns true
            1. add all cells outside of the sense range to nonLeakTiles.
        ii. else //sense returns false
            1. add all cells inside of the sense range to nonLeakTiles
        iii. bfsNotInSet(ship, nonLeakTiles, botPath, botPosition)
            1. //Updates botPath with the shortest path starting from botPosition where the goal nodes are any open tile that's not a member of nonLeakTiles
    b. else //bot path has elements in it, thus the bot is on its way to the closest open tile that could contain the leak
        i. botPosition = botPath.remove(0)
4. setNoLeakOnBot
    a. Called from experimentController if the cell the bot moved to wasn't the leak.
    b. Adds botPosition to nonLeakTiles

Bot 2

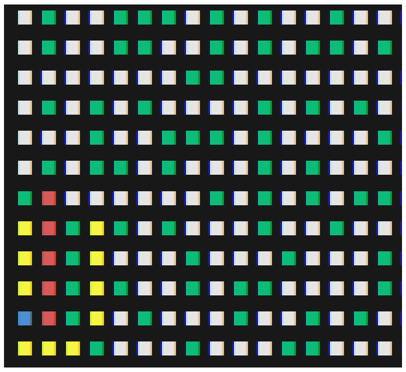
1. Overview
    a. Bot 2 is a subclass of Bot 1 so it shares its functions. It utilizes nonLeakTiles and Bot 1 methods to process information from a sense.
    b. Bot 2 uses the same sense method as Bot 1 but attempts to use it more effectively by utilizing its full range.
    c. Bot 1 tends to "waste" some of its detection range. This occurs in 3 ways:

i. The bot senses when close to the edge of the ship. The range of the sense that goes off the edge of the ship could be used if the bot was further from the edge.

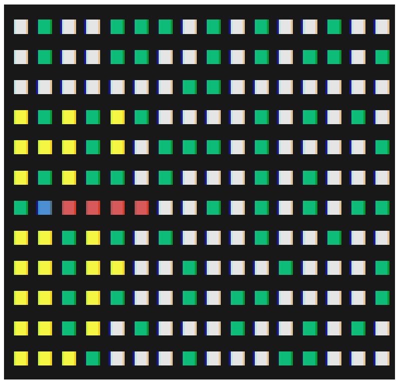
   1. **In examples: White = open cell, Green = closed cell, blue = bot. red = botPath, yellow = nonLeakTile**
   2. 

      Bot spawns at the edge of the ship

ii. The bot senses close to where it already sensed. In bot 1's algorithm, the bot repeats: sense, then moves to the nearest open cell possibly containing the leak. Often the closest tile will be directly outside the sense range, meaning the new sense will overlap with the old sense's range.

   1. 

      Bot has an initial scan and proceeds to the nearest open cell

      The bot moves to the end of the path and then scans. Notice how the tiles below the bot were already scanned.

iii. The bot keeps sensing after the first sense returns true. The first sense returning true means the leak has to be within the sense range. Bot 1 continues to sense even though as long as it stays in that range it will always return true.

2. Strategy
   a. After the ship is generated, select optimal sensing locations throughout the ship and store them in a set. Start the bot in "sensing mode" where it repeatedly moves to the nearest sensing location then senses.
   b. Once a sense returns true, have the bot exit "sensing mode" and have it solely explore the sense range.
3. Optimal Sensing Locations

   a. Populating the set of sensingLocations is done by the method setSenseLocations.
   b. Process
      i. The program divides the shipEdgeLength by 2k+1. This represents how many full senses can fit into the ship.
      ii. The program loops through the ship column by row adding the tile ship[row,col] to the senseLocations set. At the end of each loop, it adds 2k+1 to row or col based on which loop was completed.
   c. What if the ship doesn't divide evenly?
      i. If shipEdgeLength % 2k+1 does not equal 0, the program loops an extra time for each column and row. Instead of adding 2k+1, it adds k so as to not put the bot out of bounds.
   d. getOpenNeighborsIfClosed
      i. This function takes in a tile, and if it's closed, will return an open neighbor of the tile if it has one.
      ii. This function is used on the tile being added to senseLocations because a closed tile could be added. In that case, a random open neighbor of the original tile will be added to senseLocations. If there are no open neighbors, that tile is skipped.

4. Sometimes Misses
   a. Since the senseLocations are sometimes misplaced or even missing due to being placed on closed cells, the ship won’t be fully explored solely from sensing at senseLocations.
   b. Thus, botTwo will switch to bot 1s algorithm if senseLocations is exhausted without finding the leak.
5. botAction
   a. if(first sense has not been found) //bot still in “sensing mode”
      i. if(senseLocations not empty)
         1. if(botPath is empty and senseLocations contains botPosition) //bot is not moving towards a cell and the bot is on top of the sense location
            a. senseLocations.remove(botPosition)
            b. if(sense)
               i. add all cells outside of sense range to nonLeakTiles
               ii. set first sense found to true
            c. else
               i. add all cells inside sense range to nonLeakTiles
               ii. bfsInSet(ship, senseLocations, botPath, botPosition)
                  1. //bfs with goal cells being in the set sense Location. Updates botPath with the path to the closest senseLocation

            2. else if (botPath is empty) //Handles the start since botPath will be empty but the bot won’t be on a senseLocation
                a. bfsInSet(ship, senseLocations, botPath, botPosition)
                b. botPosition = botPath.remove(0)
            3. else //bot has a path already
                a. botPosition = botPath.remove(0)
        ii. else //senseLocations is empty and no leak has been found, switch to Bot 1 algorithm
            1. Identical to Bot 1 algorithm except that when sense returns true, set first sense found to true. This takes the bot out of sensing mode even if it switched to Bot 1’s algorithm
    b. else //The first sense has been found
        i. if(bot path is empty)
            1. bfsNotInSet(ship, nonLeakTiles, botPath, botPosition
        ii. botPosition = botPath.remove(0)
6. setNoLeakOnBot
    a. Identical to Bot 1

Bot 3

1. Overview
    a. Bot 3 uses a probabilistic implementation of sense.
    b. Properties
        i. tileProbabilities is a hashmap
            1. key: a cell on the ship
            2. value: the probability of the cell containing the leak
        ii. highestProbabilities is a set of cells in the ship
    c. The Bot 3 constructor takes in a double, alpha to weigh the sense.
    d. Bot 3’s loop
        i. Sense, then update all elements in tileProbabilities based on the reading of sense.
        ii. Plan the route to the cell with the highest probability of containing the leak. Move to the end of the route, updating probabilities when passing cells.
2. initializeProbabilities
    a. The method that initializes tileProbabilities at the start of the experiment given a ship. It is run in experimentController’s setBotThree method.
    b. It counts the number of open tiles in the ship then loops through each cell setting its probability to 0 if it's closed and (1/number of open tiles) if it's open.
3. sense
    a. The probability of sense returning true is $e^{(-\alpha(d-1))}$
        i. The distanceToTile method finds the distance between two tiles using bfsInSet where the goalSet is made up of just the tile being searched for. It returns the length of the list bfsInSet updates.

b. Once the probability of the sense succeeding is determined, sense uses a randomized probability roll between 0-1 and returns if the probability of the sense is greater than the random probability.

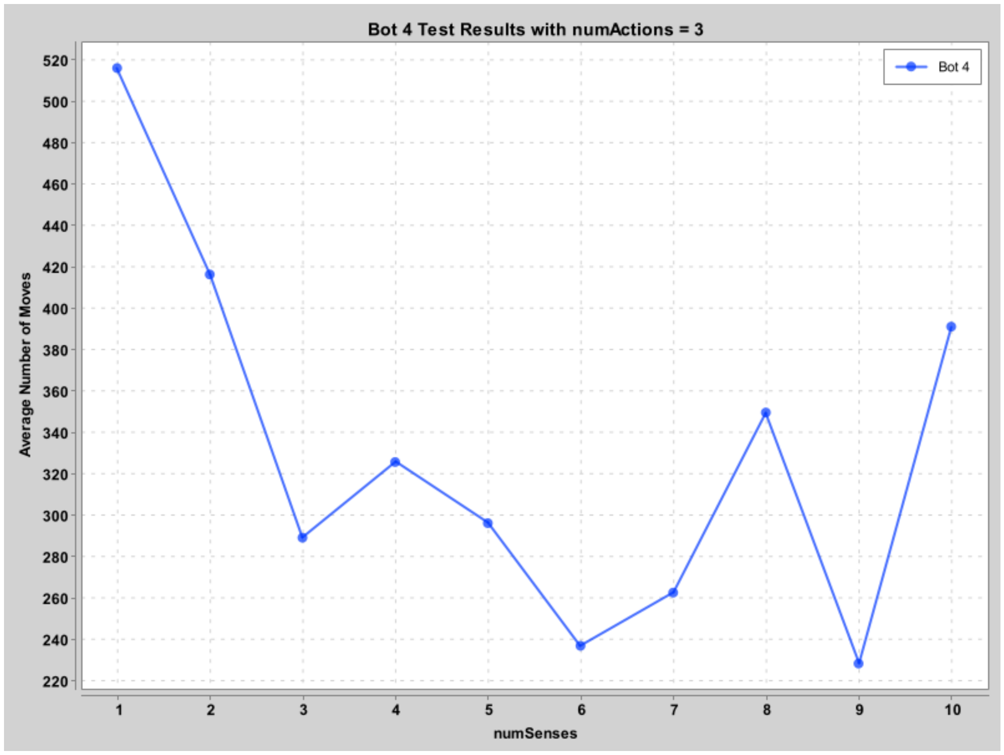
4. updateProbailitiesFromSense
    a. Takes in whether there was a beep as a parameter and updates the probability of all cells based on the sense result given.
    b. Calculations
        i. Beep
            1. For all cells in the ship j, their probability will be recalculated as P(leak in j | beep in cell i)
            2. = P(beep in cell i | leak in j) * P(leak in j) / P(beep in i)
                a. P(beep in cell i | leak in j)
                    i. The sense formula is applied here.
                    ii. = e ^ (-α (distance(i , j)-1))
                b. P(leak in j)
                    i. Stored in tileProbabilities
                c. P(beep in i)
                    i. Marginalize leak location over all cells.
                    ii. =P(beep in i and leak in i) + … + P(beep in i and leak in m)
                    iii. = P(leak in i)P(beep in i | leak in i) + … + P(leak in m)P(beep in i|leak in m)
                        1. P(leak in i) … P(leak in m)
                            a. stored in tileProbabilities
                        2. P(beep in i | leak in i) … P(beep in i|leak in m)
                            a. Sense formula is applied
                            b. distance is between i and where leak is being considered
                    iv. P(beep in i) is the same no matter which j is being considered, thus it is calculated once outside the loop that updates all tileProbabilities.
            3. Calculate this formula and apply it to all tiles in tileProbabilities.
        ii. No Beep
            1. P(leak in j | no beep in cell i)
            2. = P(No beep in cell i | leak in j) * P(leak in j) / P(No beep in i)
                a. P(No beep in cell i | leak in j)
                    i. The sense formula is applied here
                    ii. = 1 - e ^ (-α (d(i , j)-1))
                        1. The sense returning true or false covers all possible events, thus they must add up to 1.
                b. P(leak in j)
                    i. Stored in tileProbabilities
                c. P(No beep in i)

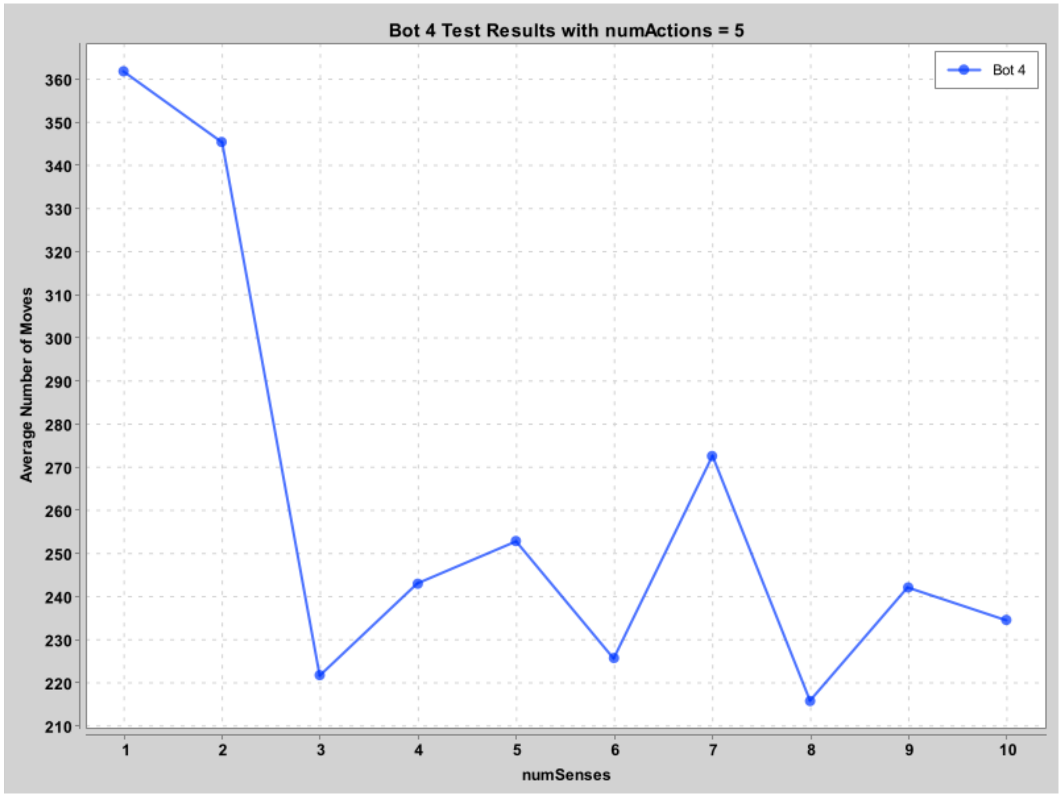
                        i. 1 - P(beep in i)
                            1. P(beep in i) + P(no beep in i) = 1

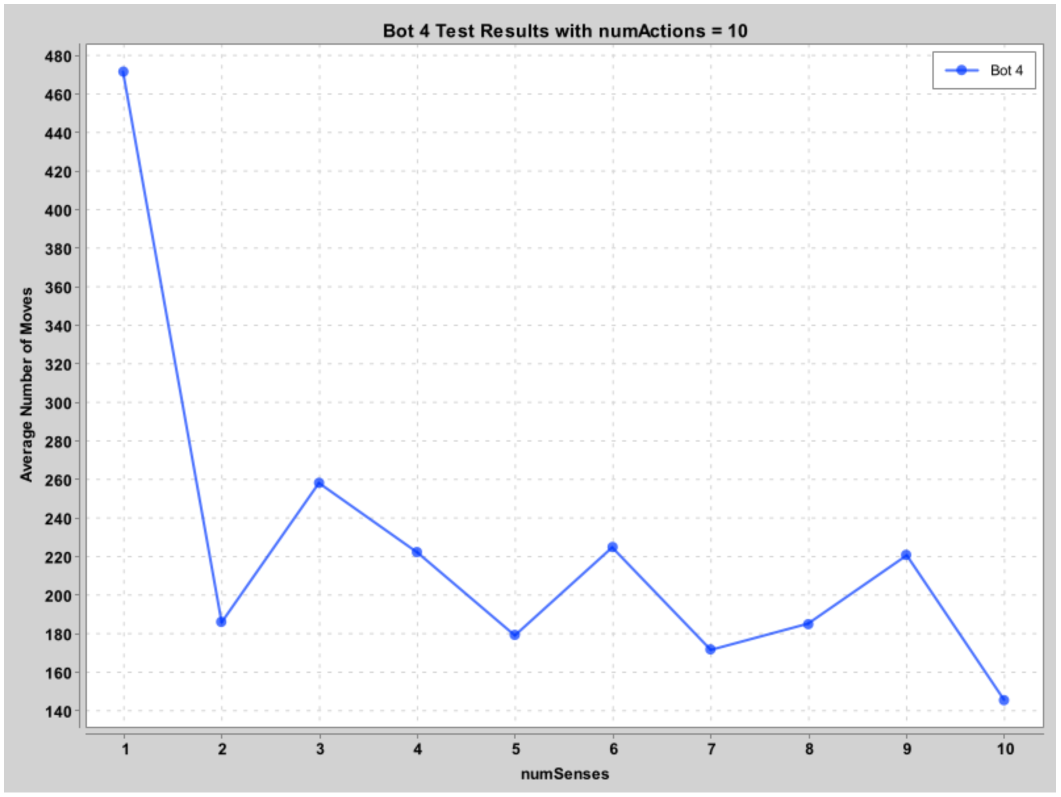
5. setNoLeakOnBot
    a. tileProb = tileProbabilities.get(botPosition)
    b. tileProbabilities.put(botPosition, 0) //leak was not found so probability is 0
    c. for each entry in tileProbabilities
        i. tileProbabilities.put(entry.getKey(), entry.getValue() / (1- tileProb))
            1. //Update each cell by dividing it by (1 - the probability of the tile removed)
6. updateHighestProbability
    a. This method updates the highestProbabilities set by looping through the tileProbabilities map.
    b. In the loop, if the current entry's probability is higher than the max, clear highestProbabilities, add the current entry, and update max. If the current entry's probability is equal to the max, add it to highestProbabilities. Otherwise the probability is smaller so it is ignored.
7. botAction
    a. if(botPath is empty) //a path was provided after a sense
        i. botPosition = botPath.remove(0)
    b. else
        i. if(sense)
            1. updateProbailitiesFromSense(beep = true)
        ii. else
            1. updateProbailitiesFromSense(beep = false)
        iii. updateHighestProbabilities()
        iv. bfsInSet(ship, highestProbailities, botPath, botPosition)
            1. //updates botPath with the goal set = highestProbabilities
    c. *Note:* updating the cell probabilities after setting a cell's probability to 0 is done by the setNoLeak() method which is called in ExperimentController's runExperiment.

Bot 4

1. Overview
    a. Bot 4 explores how a higher number of senses and number of movements the robot takes will affect the total number of actions in a run.
        i. Multiple senses gives a more accurate probability map at the cost of more actions.
        ii. Multiple movements before sensing again reduces the number of movements from cutting down on frequent sensing but comes at the cost of a less accurate probability map.
    b. Bot 4 is a subclass of Bot 3, sharing its methods to initialize and update the probabilities in the ship.
    c. Bot 4's constructor takes in:
        i. alpha

        1. a double weighing sense
    ii. numSenses
        1. Number of senses to perform in a row after movement has completed
    iii. numMoves
        1. Number of movements to perform after sensing is completed
2. botAction
    a. The bot senses when botPath is empty, meaning it has not been given a path to move.
        i. At this step, the bot will sense numSenses times in a row
        ii. Once it is done, it will update the highestProbabilityCells and then run bfsInSet on the set of highestProbabilityCells.
    b. If botPath isn't empty, the bot fully moves down the path
        i. The trimBotPath method sets the botPath to be at most numActions long.
3. Data Collection
    a. Tests were performed with varying combinations of numSenses and numActions to determine which combination was the best for the bot.
        i. Each test ran with alpha = 0.2
            1. The middle of the alpha values tested for the rest of the probabilistic bots.
        ii. The bot would sense (numSenses) times in a row, then move numMoves steps through botPath.
            1. If botPath size < numMoves the bot would traverse the entire path.
    b. The following tests were performed with numSenses ranging from 1 - 10 and where each test is given a numActions listed in the title.
        i. 

ii. 

iii. 

c. As the last graph shows, the bot is most effective at a high number of senses and actions.
d. Bot 4 uses numSenses = 10 numActions = 10.

Multiple Leaks

For multiple leak experiments, ExperimentController uses the runMultipleLeaksExperiment() method. It functions identically to runExperiment() except that it runs as long as the leaks are not null. In the loop, if botPosition = one of the leaks, that leak reference variable is nullified.

Bot 5

1. Overview
    a. Bot 5 uses a deterministic sense to find both of the leaks. It holds the same properties as Bot 1 but modifies some methods to process another leak.
2. Sense
    a. It loops through the sense's range in the same way as Bot 1. Instead it return true if either of the leaks are detected in the range
3. botAction
    a. Mimics Bot 1's botAction in all aspects except when sense returns true. Since there is another leak, no tiles outside of the sense can be ruled out. Thus, botAction only updates nonLeakTiles when sense returns false.

Bot 6

1. Overview
    a. Bot 6 is a subclass of Bot 5, utilizing common methods and properties such as its sense method.
    b. Bot 6 uses the same strategy as Bot 2. It traverses the ship in a "sensing mode" to use efficient senses. However, once Bot 2 found a leak, it would permanently switch out of "sensing mode".Bot 6 needs to switch back from searching for the leak to finding the other leak.
    c. exploringSense is an instance variable that represents if Bot 6 is in "sense mode". When the bot senses a leak, exploringSense is set to false. When the bot finds the first leak, exploringSense is set to true.
2. possibleSenseTiles
    a. A set which represents the tiles inside of a successful scan.
    b. This instance variable is implemented because with multiple leaks, the bot cannot exclude all cells outside of its sense range anymore. However, a successful sense still indicates that the leak has to be within the sense range. Thus, Bot 6 populates possibleSenseTiles with all open cells in its sense range that aren't members of nonLeakTiles.
3. setNoLeakOnBot

    a. This method adds the current botPosition to nonLeakTiles and removes botPosition from possibleSenseTiles
    b. This ensures that the bot won't back track when exploring a successful sense.
4. botAction
    a. Functions identically to Bot 2's but accommodates for 2 leaks by utilizing possibleSenseTiles
    b. When a sense returns true, the bot calls a method that fills possibleSenseTiles and sets exploringSense to true. Otherwise it continues to the next optimal senseLocation.
        i. Bot 6 also utilizes the search algorithm from Bot 5 in case the leak wasn't found from exploring senseLocations similarly to how Bot 2 utilizes Bot 1's in the same scenario.
    c. If exploringSense is true
        i. If the botPath is empty, use bfsinSet with the goal set being possibleSenseTiles.
        ii. If the bot finds the leak, possibleSenseTiles is cleared and exploringSense is set to false so the bot can return to broad sensing.

Bot 7

1. Overview
    a. Bot 7 uses the probabilistic sense implemented in Bot 3 to update a probability map in search of the two leaks.
    b. Bot 7 shares the same properties as Bot 3 and updates the probability map of all cells in the ship the same way. Bot 7 will continue searching after clearing the first.
2. Sense
    a. Bot 7 overrides the sense method to take in two leaks as parameters. It returns true if running the probabilistic sense algorithm on either leak returns true.

Bot 8

1. Overview
    a. Bot 8 tracks the probability of the two leaks being in any pair of cells.
    b. After a sense, Bot 8 will move towards the closest cell that is part of the pair(s) with the highest priority of containing the leaks.
    c. Uses the TilePair set that holds 2 tiles
    d. Properties:
        i. alpha
            1. Holds the weight for the probabilistic sense
        ii. tilePairDistances
            1. Hashmap
                a. key: a pair of tiles

            b. value: the distance through the ship between the tiles
    iii. tilePairProbabilities
        1. Hashmap
            a. key: a pair of tiles
            b. value: the probability of the leaks being in the 2 tiles.
    iv. highestPairProbabilities
        1. A set of TilePairs which holds the pair with the highest probability of containing the leaks.
    v. leak
        1. A reference to the first leak the bot found
        2. This property will remain null until the first leak is found by the bot

2. initializePairDistances
    a. This method populates tilePairDistances with the distance between every pair of cells on the ship.
    b. Uses a run of bfs starting on a tile without a target, fully depleting the queue. This finds the distance between the starting tile and every other tile on the ship. This is run for every tile on the ship.
3. initializePairProbabilities
    a. This method populates tilePairProbabilities by setting the probability of of each pair of open tiles to 1/ # of open tile pairs.
    b. This map is utilized in botAction to determine which cell the bot will move towards.
    c. This map is updated through the updateProbabilitiesFromSense and setNoLeakOnBot methods.
4. Sense
    a. Uses the same Sense as Bot 7
5. updateProbabilitiesFromSense
    a. The method that updates the probabilities of the tileProbabilityMap depending on whether a beep was received.
    b. P(a leak causes a beep in i | leak in cell j AND leak in cell k)
        i. From professor's notes:
        ii. = 1 - (1 - P(j does cause a beep in i | leak in cell j)) * (1 - P(k does cause a beep in i | leak in cell k))
            1. P(j does cause a beep in i | leak in cell j))
                a. The original formula: e ^ -alpha(distance
        iii. However when one leak is found, this is switched with the sense probability formula.
            1. When the first leak is found, the second leak cannot impact the probability of the beep.
        iv. Implementation in probabilityBeepInKGivenLeaksAccountingForFoundLeek
    c. P(leak in cell i and j | beep in cell k)
        i. = P(beep in cell k | leak in cell i and j) * P(leak in i and j) / P(beep in k)
            1. P(beep in cell k | leak in cell i and j)

                    a. 1 - (1 - P(j does cause a beep in k | leak in cell j)) * (1 - P(i does cause a beep in k | leak in cell i))
                        i. P(j does cause a beep in k | leak in cell j)
                            1. Default formula with distance(k,j)
                        ii. P(i does cause a beep in k | leak in cell i)
                            1. Default formula with distance(k,i)
                2. P(leak in i and j)
                    a. Stored in tilePairProbabilities
                3. P(beep in k)
                    a. P(beep in K and leak in pair (i,j) + … + P(beep in K and leak in pair(p,q)) //Sum of probability pairs for all unique tile pairs
                    b. P(leak in pair(i,j) * P(beep in k | leak in pair(i,j)) + … + P(leak in pair (p,q) * P(beep in k | leak in pair(p,q))
                        i. P(leak in pair(i,j)
                            1. In Hashmap
                        ii. P(beep in k | leak in pair(i,j))
                            1. 1 - (1 - P(j does cause a beep in k | leak in cell j)) * (1 - P(i does cause a beep in k | leak in cell i))
                            2. Same as before
                    c. Only needs to be calculated once before the loop through all tilePairs.
        d. No Beep
            i. P(leak in cell i and j | no beep in cell k)
                1. P(no beep in cell k | leak in cell i and j) * P(leak in i and j) / P(no beep in k)
            ii. P(leak in i and j)
                1. Stored in hashmap
            iii. P(no beep in cell k | leak in cell i and j)
                1. = 1 - P(beep in cell k | leak in cell i and j)
            iv. P(no beep in k)
                1. = 1 - P(beep in K)
            v. Since these values just need to be flipped, P(no beep in cell k | leak in cell i and j) and P(no beep in k) are calculated and if the beep argument is false, set them equal to 1 - their original value.
        e. The method runs these calculations on every pair of tile Pairs. During this process, it updates highestProbabilityPairs
            i. If the new tilePair probability is greater than the max value of the highestProbability Pair, it clears the set, updates max, then adds the tilePair.
            ii. If the new tilePair is equal to the max, it adds the tilePair.
6. setNoLeakOnBot

a. Translates the information that there is no leak on a single Tile to the map of tilePairs.
    i. If the first leak has not been found:
        1. For every tile pair in tilePairProbabilities
            a. if the tile pair contains botPosition
                i. Set the tile pair's probability to 0 in the map
                ii. Remove the tile pair from highestPairProbabilities
        2. Normalize the remaining probabilities by dividing each probability by (1 - sum of tile probabilities that were set to 0)
    ii. This part of the method runs on the intuition that if a tile is determined to not have a leak, no pair of tiles containing it could hold both leaks.

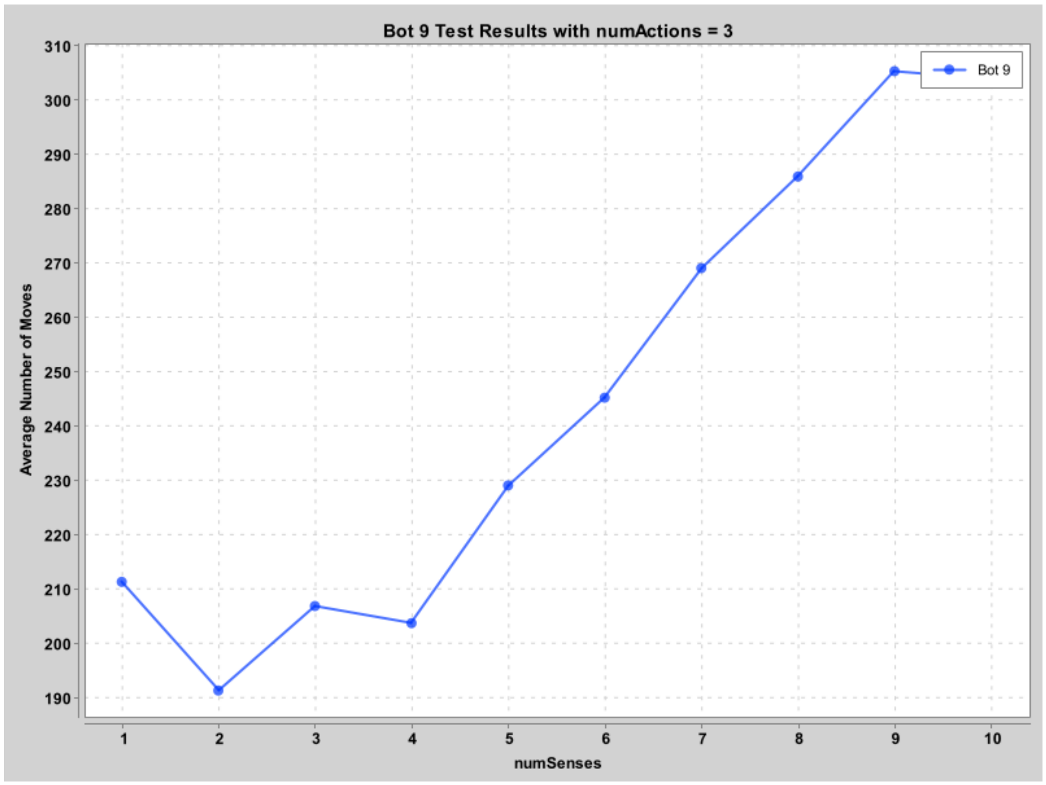
b. If the leak has already been found:
    i. When the first leak is found, the setLeakOnTile method is called from ExperimentController's runMultipleLeaksExperiment method.
        1. This method sets the leak property to the bot's Position.
        2. Then it sets the probability of all tile pairs that don't contain the leak to 0 then normalizes the map.
    ii. Now that the only tile pairs with non zero probability contain the leak, the bot can only exclude one tile pair at a time when it is traversed.

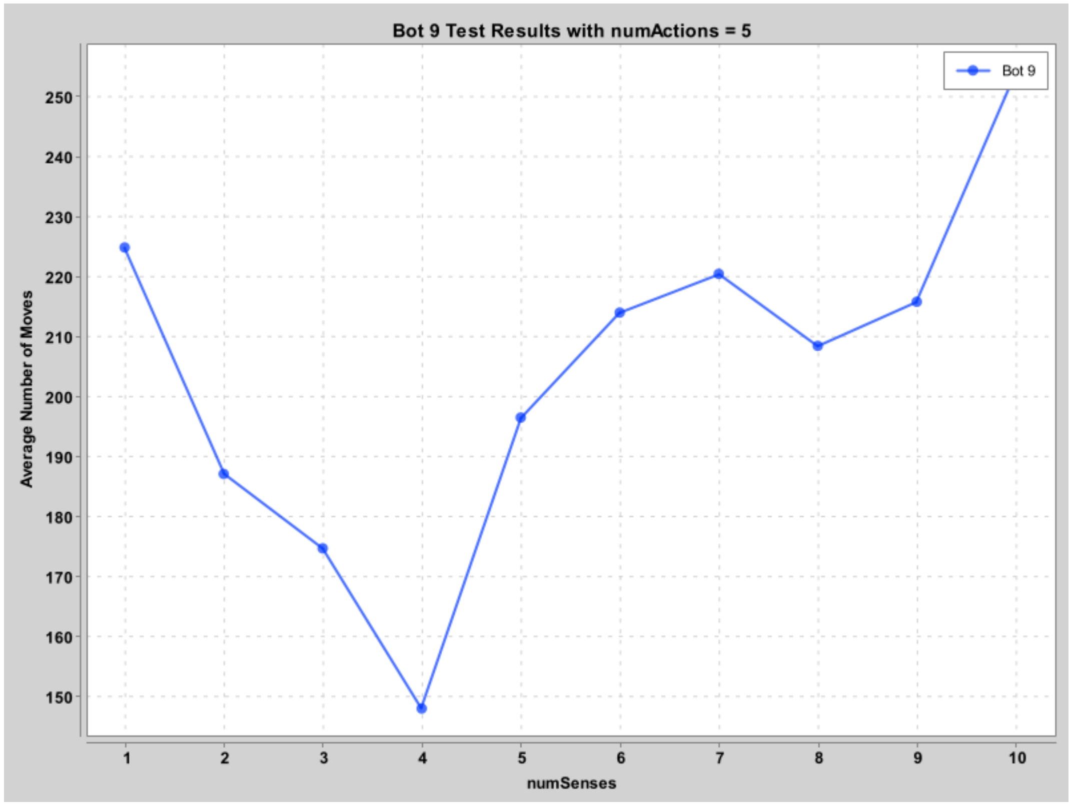
c. Pseudocode for the rest of setNoLeakOnBot
    i. else //the first leak has been found
        1. if the bot is not on the leak
                i. //This is checked because ExperimentController nullifies the leak once the bot is positioned on it, so the leak must be checked through BotEight's leak reference
            b. Set the probability of tilePair (leak, botPosition) to 0.
            c. Remove tilePair from highestPairProbability
            d. Normalize the rest of the probabilities by dividing them by 1- (original probability of tilePair(leak, botPosition).

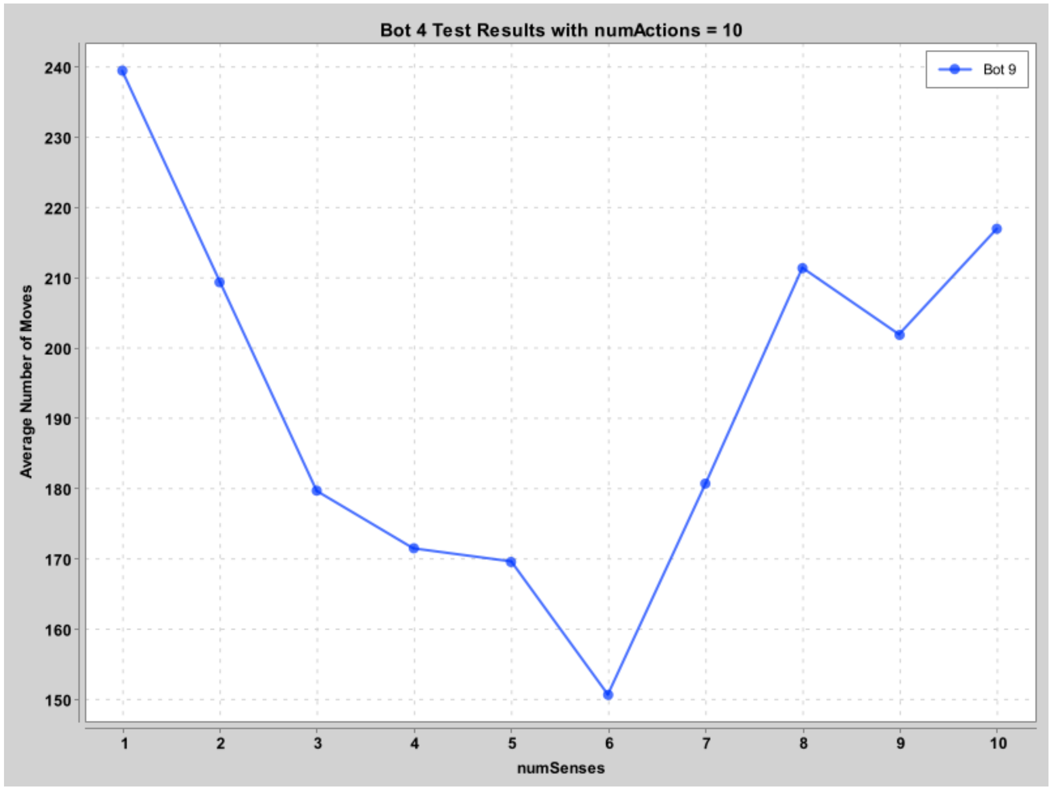
7. botAction
    a. if bot path is not empty // Bot is searching for the leak
        i. botPosition = botPath.remove(0) //Move one step in the botPath
    b. else
        i. if sense
            1. updateProbabilitiesFromSense(there was a beep)
        ii. else
            1. updateProbabilitiesFromSense(there was not a beep)
    c. create new set highestTileSet //Holds tiles for bfs to search
    d. for tilePair in highestPairProbabilities //Add all the appropriate tiles from highestPairProbabilities
        i. add the tiles in the pair that aren't equal to the botPositon or the first leak if it has already been found.
    e. Run bfsInSet with highestTileSet as the goal set.

Bot 9

1. Bot 9 uses the same probabilistic sense and probability updates as Bot 8 but tests how variations in the number of senses in a row or number of bot movements in a row will affect the number of actions.
    a. Bot 9 is a subclass of Bot 8
    b. Similar to Bot 4's numSense and numAction fields
2. Since the probability updates are calculated correctly in Bot 8, utilizing more senses to create an accurate probability map of tilePairs should speed up the bot.
3. botAction
    a. Each instance of Bot 9 contains a field for numSenses and numActions
    b. As in Bot 4, the bot would perform sense a number of times in a row equal to the numSense field. Then, rather than moving all the way to the highest probability cell pair, it would move numActions steps in the botPath then sense again.
4. Data Collection
    a. Tests were performed with varying combinations of numSenses and numActions to determine which combination was the best for the bot.
        i. As in bot 4, each test ran with alpha = <u>0.2</u>
    b.



c. 

d. 

   i. Title typo, should read Bot 9 Test Results with numActions = 10

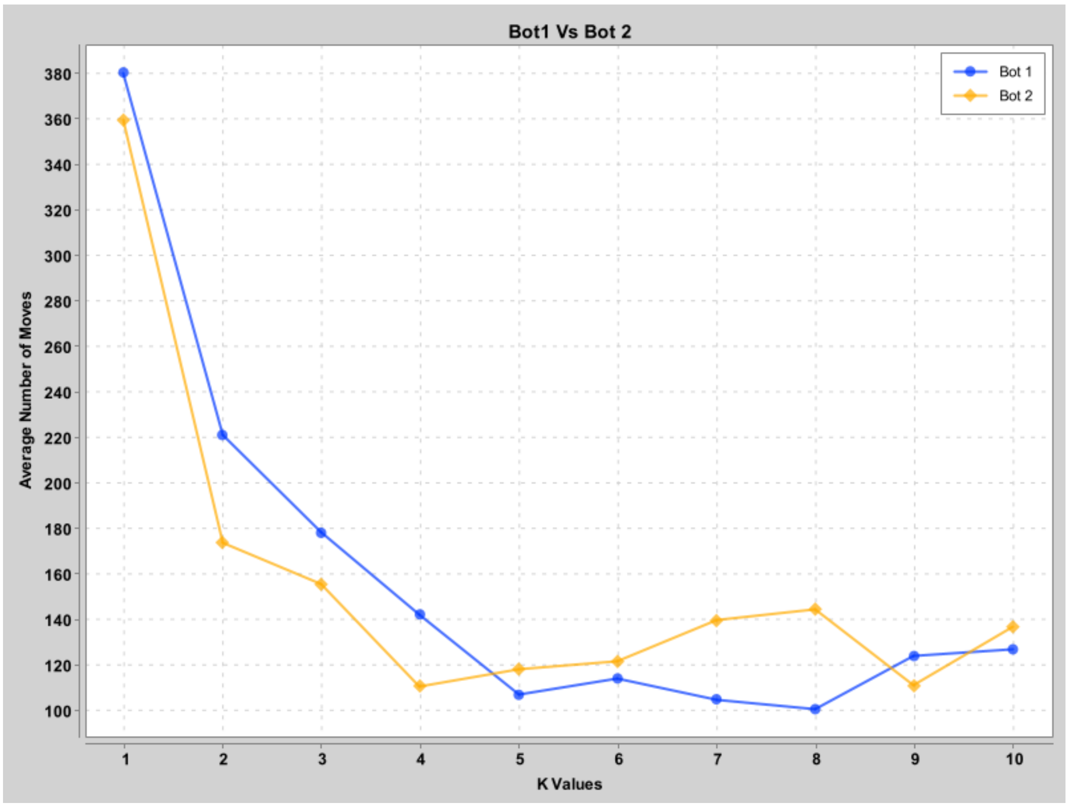
e. These results contrast Bot 4’s as the number of senses in a row was most effective at around 5-6 rather than 10.

f. Bot 9 uses numSenses = 4, numActions = 5.

Experiment Results

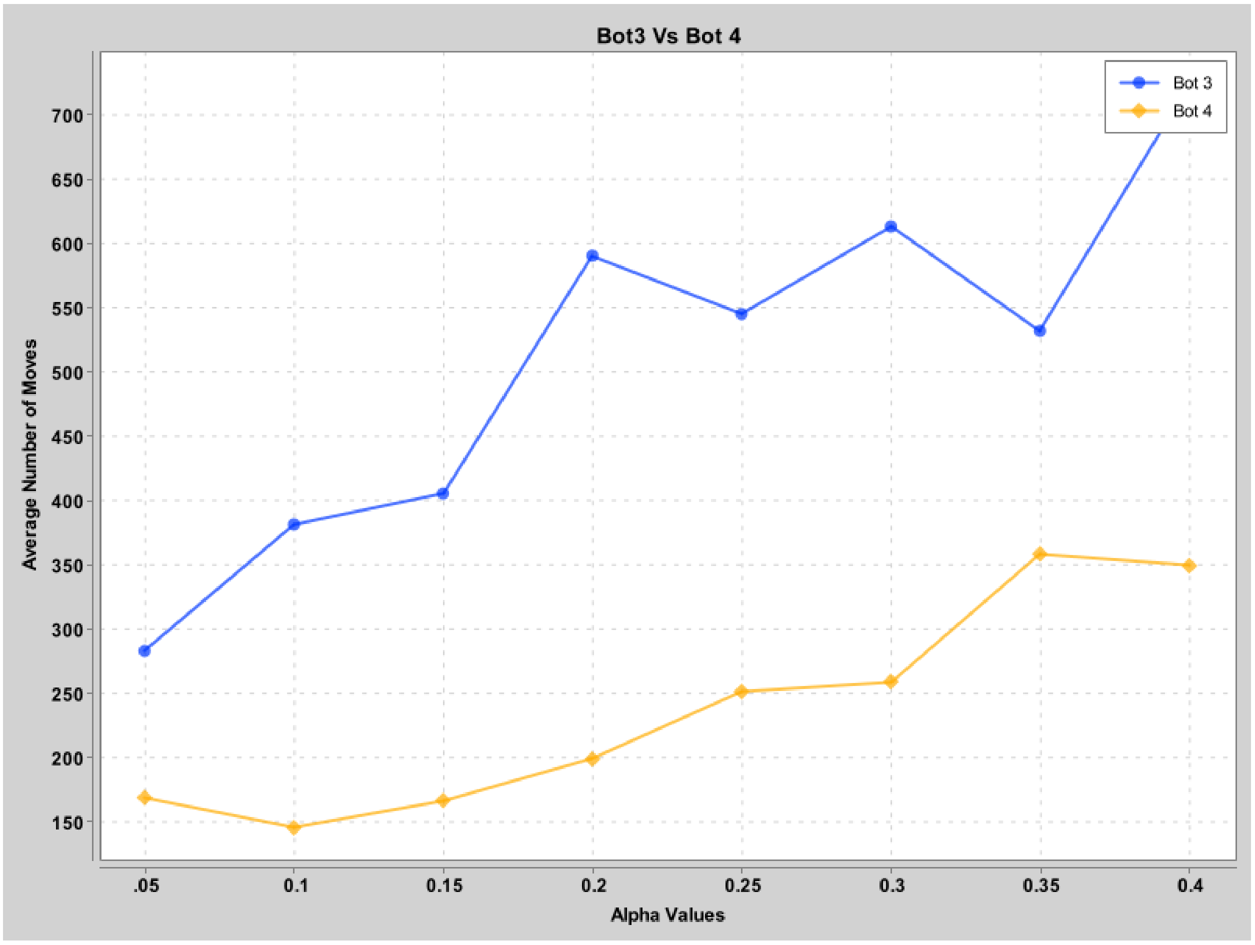
1. Overview: Testing done in the ExperimentData class
    a. The tests were performed on 30 x 30 ships with the points on the graph being averages of 50 tests.
    b. The k values for the deterministic sense ranged from 1 -10
        i. This limit for k values was chosen since a deterministic sense of k = 10 on a 30 x 30 ship would already cover most of the ship.
        ii. The lower level of k was selected to test how the bots functioned with very limited sensing capabilities.
            1. In this case of k = 1, the bot would only get notified if it was adjacent to the leak.
    c. The alpha values for the probabilistic sense ranged from 0.05 to 0.40
        i. The upper bound of 0.40 tests how the bots handle infrequent beeps. The values were capped 0.40 because testing became very lengthy for higher alpha values.
        ii. The lower bound of 0.05 tests how the bots handle frequent beeps.

Bots 1 and 2



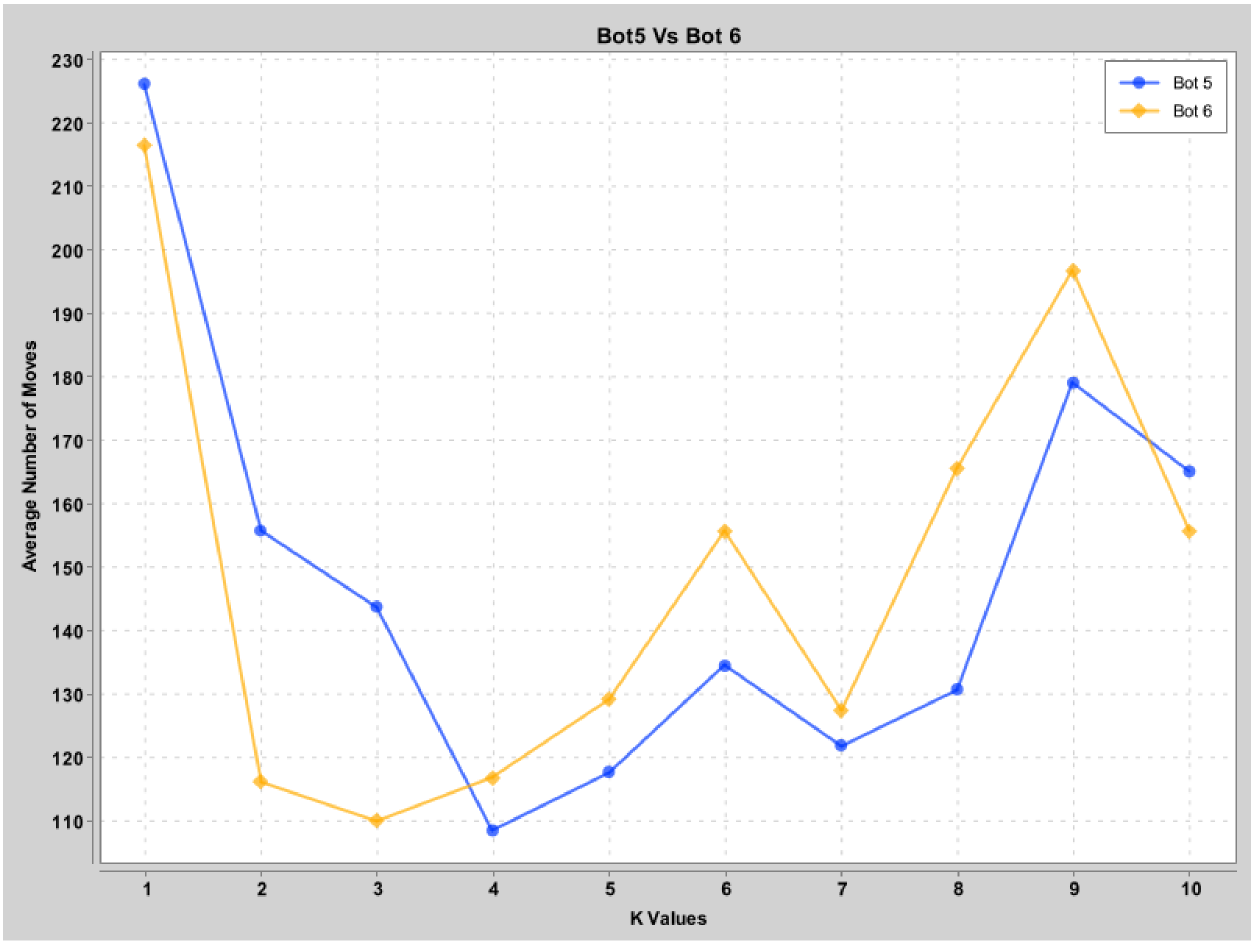
1. Bot 2 performed better than Bot 1 at lower k values but worse at higher ones.
    a. This could indicate that when the sense covers large portions of the ship, using actions to maximize the effectiveness of sense isn't as necessary.

Bots 3 and 4



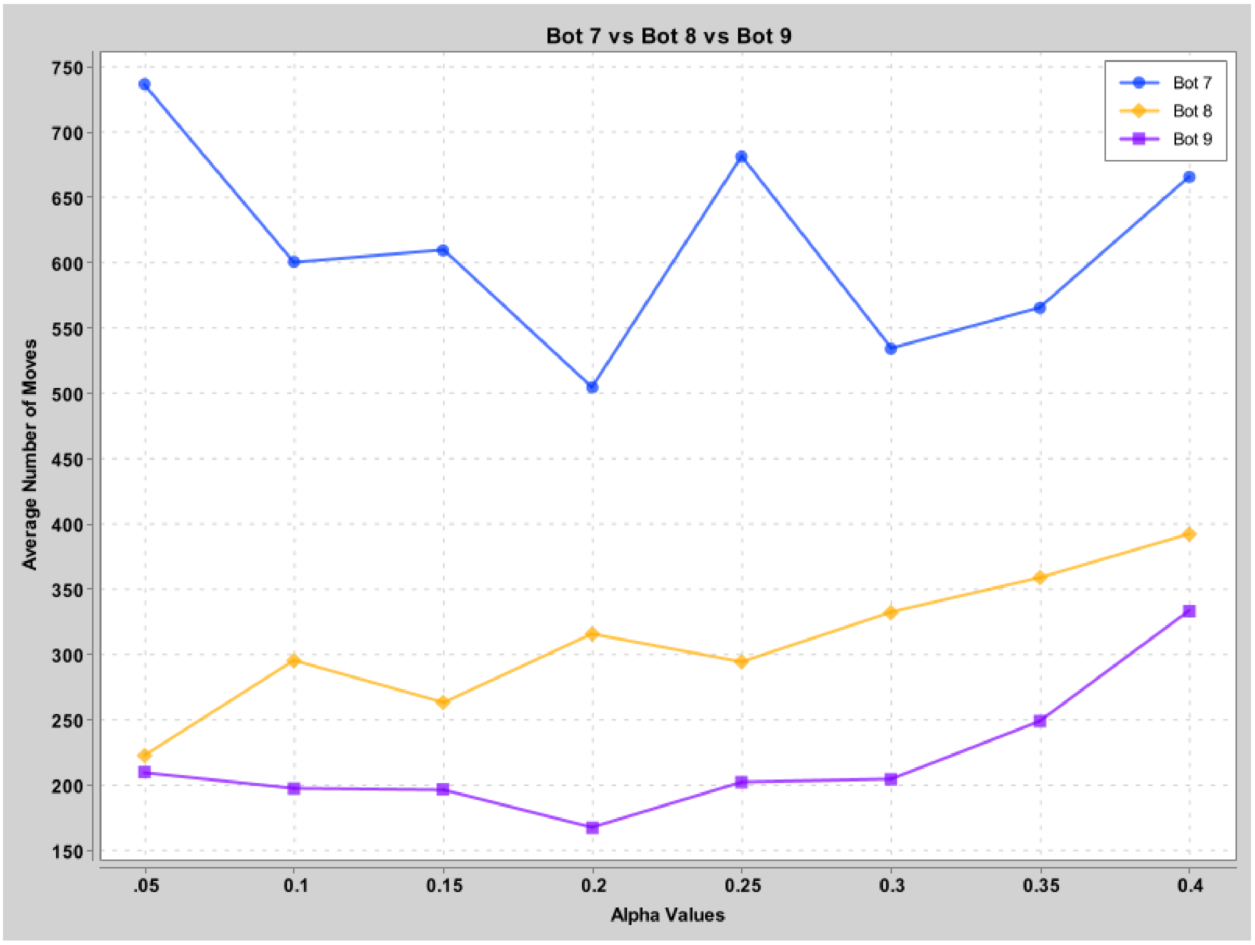
1. Bot 4 with numSenses = 10 and numActions = 10, performed better than Bot 3 at every alpha value
    a. Bot 4 performed best at alpha = 0.1, indicating that frequent beeps tend to perform better than infrequent beeps. Though there is a limit demonstrated by alpha = 0.05 performing worse than 0.1.

Bots 5 and 6



1. Similarly to the relationship between Bots 1 and 2, Bot 6 performs better than Bot 5 at low k values but worse at higher ones.

Bots 7, 8, and 9



1. The proper probability calculations of Bots 8 and 9 give it a large performance boost over Bot 7.
2. Bot 9 with numSenses = 4 and numActions = 5 performs better than Bot 8 at all alpha values, showing more senses reduce the number of actions of the bot.