# Circle of Life Lab Report

**Author: Patrick Liang – pzl4**

## Design:

## Environment:

**Graph:**

Initially planning for the Graph, I experimented with many data structures. Eventually, I realized that an ArrayList of ArrayLists is the optimal data structure to use because lookup is O(1) in time complexity and adding a new cell is also O(1) in time complexity. Even though a HashMap offers the same time complexity, there is no way to access the info of the starting cell, which is essential for this project. In addition, I decided to create a Node class for the Graph so it could store cell value and previous cell(used in BFS). Moreover, because it is an object, I decided to override the built-in equals() and hashCode() functions for this class. This is because then I could compare two different objects and see if the cell is the same. In order to do this, I imported a helper class built by Apache Commons.

Implementing the actual buildGraph() function was the core of the Graph class. I first created a function buildSkeletonGraph() which builds the initial bidirectional circular graph connecting all Nodes. I then had to build the addRandomEdges() method, which adds edges randomly which adhere to the constraints of a cell having a degree of three or less and a maximum distance of five for each connection.

**Agent:**

The constructor of this class generates a position uniformly at random for cells 0-49(size of graph).

**Prey:**

The constructor for this class generates a position on the graph that is not the Agent's cell. In addition, there is a method that chooses uniformly at random what the next move will be.

**Predator:**

The constructor for this class generates a position on the graph that is not the Agent's cell. There is a method that chooses uniformly at random what the next move will be. There is a BFS method that searches for the Agent from the perspective of the Predator. BFS is the only valid search method because it finds the shortest path, and a heuristic could not prune out search paths. In addition, it implements a getPath() method to return the path of Predator to Agent. It is used to determine the length of the path.

**Result:**

Simply three constructors are used to return data points necessary to determine each Agent's efficacy.

## Odd Agents Base Logic:

The odd Agents all follow the same pattern. After getting each neighboring cell of the Agent and each cell's distance to Prey and Predator, you compare each cell and add it to a priority list based on the rules given in the writeup. I decided to use a HashMap to store cells that belong to a qualified level. This way, when deciding which cell to travel towards, I will iterate through the HashMap starting from level one, and if there exists a cell that belongs to level one, I will choose that cell. If there are multiple cells at the first level that contains cells, I will choose uniformly at random which cell to travel towards. Afterward, Agent will move, the Prey will move at random, and the Predator will move greedily for Agents 1 and 3. In Agents 5 – 8, the Predator moves differently. Instead of always greedily moving toward the Agent, the Predator will only greedily move toward the Agent 60% of the time. The other 40% of the time, it will move uniformly at random toward one of its neighbors.

## Even Agents Base Logic:

The even Agents follow a similar pattern to the Odd Agents. However, instead of following the writeup's steps to determine each cell's viability, I created my utility function to determine the value of each cell. This utility function, called bestCell(), first evaluates the average distance from each Agent's neighbors to the Preys neighbors. For example, if Agent neighbors are 1, 2, and 3 and Prey neighbors, including itself, are 9, 10, 11, and 12, then the distance of 1 to 9, 1 to 10, 1 to 11, and 1 to 12 will all be averaged out and stored. The function does this because I appreciate the closest distance to the potential future of the Prey more than the distance between the Agent and where the Prey is currently. I also find the Predator distance between it and each Agent neighbor in the same calculation. I then get the average distance for some cells and subtract it from the distance of the Predator to the Agent. This calculation is my base utility.

I iterate through Prey distances to the Agent, including itself, and Predator distances to the Agent. When the shortest distance is found to the Prey, I update the utility value for that cell by adding 75*(0.5/(number of Prey neighbors, including itself)). When the shortest distance is found to the Predator, I update the utility value for that cell by adding 100*(-0.5/(number of Predator neighbors, including itself)). The values I use are arbitrarily found through experimentation. However, I did choose the shortest distance to the Predator to be weighted more heavily because I value survivability over attempting to win greedily.

After all utility values are calculated, I choose the utility with the greatest value when it is greater than zero. If the greatest utility value is less than or equal to zero, I opt to move away from the Predator.

## Probability Distribution Logic:

To keep a valid belief vector, there exist constraints that need to be satisfied. The first is that the entire belief vector should add up to 1.0. Secondly, when a cell has been surveyed not to contain the Predator/Prey, the other belief updates must be updated in accordance with Bayes' theorem.

Bayes' theorem:

P(B|A) = $\frac{(P(A|B)*P(B))}{P(A)}$ -> $\frac{(1*P(B))}{P(A)}$; P(B) is $1/_N$, N is number of ties; the length of the shortest path dictates ties; P(A) is the calculated chance of the current node. Simplified, it becomes $\frac{1}{(1-(P(B))}$.

After Bayes' theorem gets applied to the belief vector, normalization is required to bring down the scale of each node to lie between 0 and 1. The normalizing constant, in this case, would be the sum of the belief vector immediately after Bayes' rule is applied.

Next, to find the probability of Prey/Predator after moving, I used a Markov Chain to update the belief vector.

The Prey transition matrix is filled with either 0 or the random probability that the Prey would move into the cell. For example, if a Prey has three neighbors, the row in that transition matrix would have ¼ for each cell that qualifies as a move for the Prey.

The Predator has two transition matrices. The first one is nearly identical to the Prey transition matrix. Instead of considering itself as a move, it only considers its neighbors.

The second Predator transition matrix is determined by the shortest path to the Agent's current position. The denominator for each cell is the count of shortest paths from its current position. For example, if two cells are tied with the shortest path, then the probability would be ½ for each cell.

The Prey belief vector would be updated using matrix multiplication with the Prey transition matrix.

The Predator belief vector would first be cloned into two separate vectors. The first would be updated using matrix multiplication with the Predator random transition matrix. The second one would be updated using matrix multiplication with the Predator greedy transition matrix. The first one would then be multiplied by 0.6 because there is a 60% chance that the Predator would move greedily. In the end, add clone1 and clone2 to get the final Predator belief vector. The first one would then be multiplied by 0.4 because there is a 40% chance that the Predator moves uniformly at random. In the end, add the two vectors for the final Predator belief vector.

This completes the Markov Chain.

## Complete Information:

### Agent One:

There is not much to add to this Agent, as it follows the Odd Agents' basic logic. It starts off knowing both Prey and predator location.

### Agent Two:

Just like Agent One, this follows the Even Agents' basic logic. It starts off knowing both Prey and predator location.

**Helper Methods:**

<u>searchPred():</u> Traditional BFS to search for Predator distance based on agent location

<u>searchPrey():</u> Traditional BFS to search for Prey distance based on agent location

Even though they are the same code, I opted to make two different methods, so it is easier to know what each BFS is used for when reading the code.

## Partial Prey:

**Agent Three:**

It starts only knowing where the Predator is. In addition to the basic logic, it is necessary to update beliefs and survey the location of the Prey. In order to get the best survey, I survey the cell with the highest probability of containing the Prey and choose uniformly at random from the selected cells if there are ties.

To keep a valid belief vector, there exist constraints that need to be satisfied. The first is that the entire belief vector should add up to 1.0. Secondly, when a cell has been surveyed not to contain the Prey, update the belief vector accordingly, then normalize. Lastly, a Markov Chain is used to apply the probability for the next time step accurately.

**Agent Four:**

Same as Agent Three, except it uses the bestCell() function to determine the utility of cells instead of the provided rules.

Agent Three and Four performs all these operations through the following helper methods:

<u>updateProbability():</u>

Applies Bayes' theorem.

<u>maxIndex():</u>

Returns the index with the greatest belief.

<u>maxBelief():</u>

Returns greatest belief value from beliefs vector.

<u>initialBelief():</u>

Initializes belief vector with $\frac{1}{49}$ for every cell except the Agent's.

<u>beliefSum():</u>

Returns the sum of the belief vector.

<u>randomSurvey():</u>

Chooses the most probabilistic cell and adds it to an ArrayList. Then, chooses uniformly at random between tied most probabilistic cell.

matmul():

Performs matrix multiplication using dot product.

dotProduct():

Dot product with belief vector as a row, not a column.

normalize():

Normalizes the belief vector.

## Partial Predator:

### Agent Five:

It starts with knowing where both Predator and Prey are but does not know where the Predator moves to. Utilizes surveying and Markov Chain to update beliefs.

The main difference in surveying between Partial Prey and Partial Predator is that tie breaks are now by the shortest path instead of breaking up ties uniformly at random. If there are multiple shortest paths, the tiebreak becomes uniformly random.

Another difference would be that the Partial Predator has to keep updating the second transition matrix after each step the Agent takes. This is because the second transition matrix is greedily built, not randomly.

### Agent Six:

Same as Agent Five, except it uses the bestCell() function to determine the utility of cells instead of the provided rules.

### Notable helper method(s):

updateTransMatrix:

Takes agent position and the graph and fills out the transition matrix with greedy transitions. Utilizes BFS search for shortest paths.

## Combined Partial Information:

### Agent Seven:

Starts off knowing where Predator is, but not Prey. In addition, does not know where Predator is moving with 100% confidence.

A notable difference is that now I need to maintain two separate belief vectors, one for Prey and one for Predator. Otherwise, nothing new to add.

### Agent Eight:

Same as Agent Seven, except it uses the bestCell() function to determine the utility of cells instead of the provided rules.

## Faulty Combined Partial Information:

### Agent Seven Faulty:

The only difference is that now the survey is 10% faulty. This means the survey will ping a false negative 10% of the time.

### Agent Eight Faulty:

Same as Agent Seven Faulty, except it uses the bestCell() function to determine the utility of cells instead of the provided rules.

## Fixed Faulty Combined Partial Information:

### Agent Seven Faulty Fixed:

Belief updates are now updated. Instead of the normal Bayes' rule, updates by making the current surveyed node 0.1 of the original and normalizing the belief vector.

### Agent Eight Faulty Fixed:

Same as Agent Seven Faulty, except it uses the bestCell() function to determine the utility of cells instead of the provided rules.

## Data Collecting:

I generate 30 different graphs in the main method and run all agents on the same graph 100 times. This results in 3000 simulations in total. I collect the data through the Results class and store it in a .txt file.

In addition, the hung constraint is 5000 steps, which never occurs.

## Grey Box Question:

1. The maximum amount of edges for 50 vertices and a degree of three for each vertex, thus the maximum number of edges possible, is 75. Because the initial graph is 50 vertices with a degree of two, the initial graph already has 50 edges. Thus, the greatest amount of additional edges added possible is 25, 75 minus 50.
   The minimum amount of additional edges is 18. Because the rules do not allow for duplicates, consecutive edges cannot have a third edge between them. So if we start at edges zero and one, we want a distance of five from both sides to be completely occupied and for the next pair of consecutive edges to be six away from the second point(e.g., pair(0, 1), next pair would start six away from one, (7, 8)). This means that for a graph with 50 vertices and a maximum degree of three for each vertex, the maximum number of consecutive pairs is seven, allowing for 18 additional edges to be added.

2. Updating beliefs by move/survey: Use Bayes' theorem to propagate beliefs. When a cell of probability 1/49 is found not to contain the Prey/Predator, it is removed from the denominator of every other cell(e.g., 1-(49-(1/49)). Afterward, normalize by dividing each cell by the sum of the entire belief vector.
   Transitioning states: Use a transition matrix that contains the probability of Prey/Predator moving to that cell from the current cell(e.g., Prey is at cell 0, neighbors are 1, 2, 3, there is a ¼ chance for the Prey to move into each cell). Multiply this transition matrix with the belief vector to complete the Markov chain.

3. There are many differences between the probability updates from the Prey to the Predator. The main difference is how the transition updates happen. The Predator requires two transition matrices. The first is similar to the Prey's transition matrix, except that the denominator will be strictly neighbors, not including itself. After the matrix multiplication, multiply a cloned belief vector, clone1, by 0.4 because there is a 40% chance that the Predator moves randomly. The second transition matrix is in the same vein as the first transition matrix, where each cell is determined by the likelihood of the Prey/Predator moving toward it. Instead of uniformly at random, the Predator moves towards the cell closest to the Agent. This means there is a lot less probability and, instead, is typically one or two cells, which means more certainty. After the matrix multiplication, multiply a cloned belief vector, clone2, by 0.6 because there is a 60% chance that the Predator will move greedily. In the end, add clone1 and clone2 together to get the final Predator belief vector.
   1. The Agent should move toward the cell with the most "success."
   2. The node the Agent should survey is the node with the highest probability of winning for the Agent.
   3. The Agent could use a utility function to calculate the utility of each cell.

## Data:

| | Survival Rate(%) | Predator Catches Agent(%) | Agent Catches Prey(%) | Agent Runs into Predator(%) | Simulation Hang(%) | Prey Survey(%) | Predator Survey(%) |
|---|---|---|---|---|---|---|---|
| **Agent One** | 89.80 | 10.20 | 59.70 | 0.00 | 0 | N/A | N/A |
| **Agent Two** | 95.06 | 4.93 | 61.83 | 0.00 | 0 | N/A | N/A |
| **Agent Three** | 83.83 | 16.17 | 52.17 | 0.00 | 0 | 6.52 | N/A |
| **Agent Four** | 87.2 | 12.80 | 51.47 | 0.00 | 0 | 5.44 | N/A |
| **Agent Five** | 81.93 | 15.83 | 53.77 | 2.23 | 0 | N/A | 66.90 |
| **Agent Six** | 85.53 | 12.37 | 51.87 | 2.10 | 0 | N/A | 65.67 |
| **Agent Seven** | 79.2 | 18.07 | 46.93 | 2.55 | 0 | 1.81 | 52.39 |
| **Agent Eight** | 84.93 | 13.27 | 51.80 | 1.80 | 0 | 1.71 | 50.25 |
| **Agent Seven Faulty** | 55.23 | 29.33 | 32.03 | 15.43 | 0 | 1.87 | 34.36 |
| **Agent Eight Faulty** | 69.40 | 25.1 | 42.00 | 5.50 | 0 | 1.82 | 36.29 |
| **Agent Seven Faulty Fixed** | 67.07 | 21.90 | 40.00 | 7.83 | 0 | 1.62 | 37.63 |
| **Agent Eight Faulty Fixed** | 77.47 | 17.10 | 48.00 | 5.43 | 0 | 1.57 | 39.18 |

**Notes:**

**Survival Rate:** how often the Agent catches the Prey or Prey runs into Agent

**Predator Catches Agent(%):** On average, out of 100 simulations, the Predator catches the Agent

**Agent Catches the Prey(%):** On average, out of 100 simulations, the Prey runs into Agent

**Agent Runs into Predator(%):** On average, out of 100 simulations, the Agent runs into Predator

**Simulation Hang(%):** How often the Agent would be hung

**Prey/Predator Survey(%):** How often the Prey/Predator survey would be correct

## Analysis:

The Agents that I design always have a better success rate than the specified Agents. What it lacks, however, is the survey rating success and the Agent catching the Prey success. In contrast, the Predator catching the Agent is always lower for my Agent than the specified Agent. The reason my Agents outperform the specified Agent is in the data. The Predator always catches the Agent fewer times. This is because when designing the utility function, I emphasized running away from the Predator more than running to the Prey. I believe that by valuing surviving over winning, the Agent becomes more successful overall.

Even though the success rate of my Agents is always better than the success rate of the specified Agents, I do not think that my Agents utilize the available information as effectively as possible. I believe that this is the case because I do not utilize probabilities at all when I am calculating utility. I am always assuming that the cell with the highest probability of containing the prey/predator contains the prey/Predator. This means that I might be heading away/towards the wrong cell in the graph, leading to less capturing success. That is why on average, the Agent does not capture the Prey more frequently when comparing my Agents and the specified Agents.

When comparing the different information settings, it is clear that Complete Information does the best, Partial Prey the second best, Partial Predator does the third best, and Combined Partial Information does the worst. It is easy to see why Complete Information does the best, it has all the information. Partial Prey versus Partial Predator, however, is not due to a difference in quantity of information known, it is a difference in quality. In order for an Agent to win either the Prey has to run into the Agent, or the Agent has to catch the Prey. This would make it seem that Partial Prey information is of lower quality. This is wrong. Not knowing the location of the Predator is what causes the success rate to fail. As seen in the data table, the percentage of Agent running into the Predator starts happening once Predator location is not known. This increases the death rate of the Agent. Similar things happen in Partial Prey, except when Prey runs into the Agent on accident, it is always beneficial. Therefore, the Partial Predator setting has a lower success rate than Partial Prey setting due to the quality of information known.

When the Combined Partial Information setting survey is faulty, Agents Seven and Eight take a nosedive in all statistics except the prey survey. The success rate drops by 24 in Agent Seven and a drop of 15 for Agent Eight. However, once I update the belief updates, the success rate reaches a median between faulty Agents Seven and Eight and normal Agents Seven and Eight, with an increase of 12 for Agent Seven and an increase of 8 for Agent Eight.

I updated the belief system by opting not to remove the surveyed probability. Instead, I multiply it by 0.1 because of the 10% that it is faulty. I then normalize the belief vector to update the probabilities. Normalizing has the same effect as Bayes Rule because the new sum considers the 10% false negative rate.

To theoretically improve upon a faulty survey, I could opt to survey the same spot multiple times to confirm the veracity of the survey, the more surveys, the more accurate.