Project Report - II

Circle of Life

Sri Krishna Kaashyap Madabhushi
Net ID: sm2599

Lokesh Kodavati
Net ID: bk576

August 24, 2023

Contents

1 Implementation

1.1 Environment

For this project, we have represented our graph as an adjacency matrix. As a precomputation step, we have executed Floyd Warshall algorithm on our graph such that we have a distance matrix that stores distances from every pair of nodes. The following are the steps we used to generate the graph

1 Initially we defined an adjacency matrix of size *50 x 50*.

2 After creating the matrix, we iterated through every node adding an edge between all the adjacent nodes.

3 After this step has been performed, all the nodes in the graphs had the degree 2.

4 To further connect the graph, if any node has a degree 2, we have selected a random node in its [-5, +5] distance range and added an edge if the degree of both the nodes are 2.

5 We did this by iterating through every node.

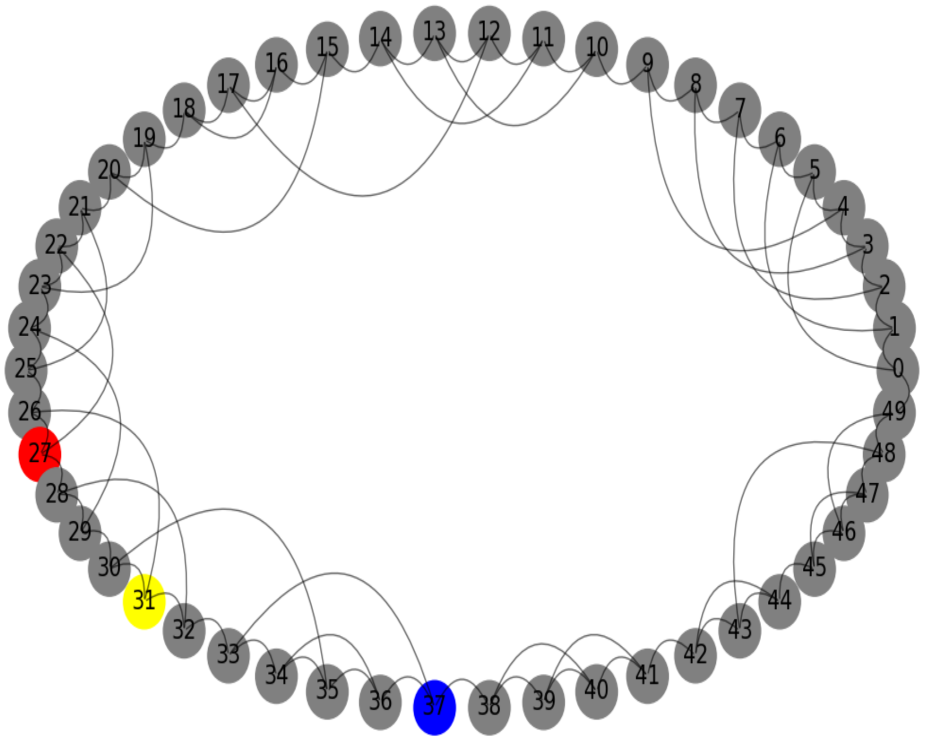


Figure 1: Our generated Environment where red node is the Predator, yellow node is the prey and the blue node is the Agent

With this setup, you can add at most 25 additional edges (why?). Are youalways able to add this many? If not, what's the smallest number of edges you're always able to add?

According to graph theory; sum(Degree) = 2 * (E) where degree is the degree of all the nodes and E is the number of edges. Therefore, in the given setup, we have 50 nodes with the degree of 2. The maximum number of edges that are possible using the above property is 50. If we assume that we get the degree as 3 for all the nodes, then the maximum number of edges is 75. Since we have already added 50 edges, the maximum we can add is 25. On contrary, using our code to add edges in the range [-5, 5], the least amount of edges we can add is 18 (By ignoring the middle nodes when considering the nodes in the batches of 5).

Now, this environment is occupied by three entities:

- **Prey:** We store the prey's position in a variable. At every timestep, we select a random choice from a list of choices containing the current position and its neighbours. We simply change the prey's value to the new value based on the choice.
- **Predator:** Similar to the prey, we have stored the predator in a variable. Since we have already precomputed the distance from all nodes using the Floyd Warshall algorithm, we look at the predator's neighbours' that have the minimum distance and select that node as its next movement. If there are multiple nodes with the same distance, we break ties at random.
- **Agent:** Finally, we store the agent's current position in a variable and follow the environment rules to travel in the graph.

2 Precomputation

As a precomputation, we have performed the Floyd Warshall algorithm on the graph. The main idea of Floyd Warshall is to compute the distance from every node to every other node in the graph. This distance metric is used in all our distance calculations throughout the project.

3 Agent 1

In this setting, the agent exactly knows where the prey and predator are at all times. Therefore, we have implemented this agent using the given rules that prioritize moving closer to prey and further from the predator. Since we have already pre-computed the distance matrix using Floyd Warshall algorithm, our distance calculations take constant time and therefore we take informed decisions quickly. At each time step, we try to check the distance using our matrix and try moving closer to the prey using the following rules:

- Neighbors that are closer to the Prey and farther from the Predator.
- Neighbors that are closer to the Prey and not closer to the Predator.
- Neighbors that are not farther from the Prey and farther from the Predator.
- Neighbors that are not farther from the Prey and not closer to the Predator.
- Neighbors that are farther from the Predator.
- Neighbors that are not closer to the Predator.
- Sit still and pray

3.1 Performance and Survivability

We have executed this agent for 30 new graphs, 100 times each and have observed that on an average, the agent catches prey **90%** of the times. The following graphs signify the success rate for each graph on a scale of 100 since we run 100 times on each graph.

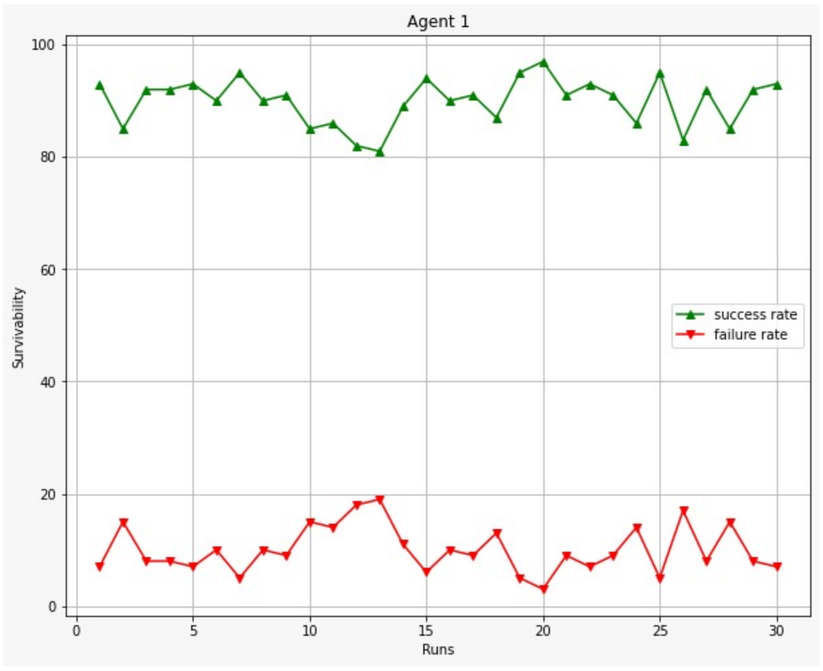


Figure 2: Agent 1 Success Rate vs Failure Rate

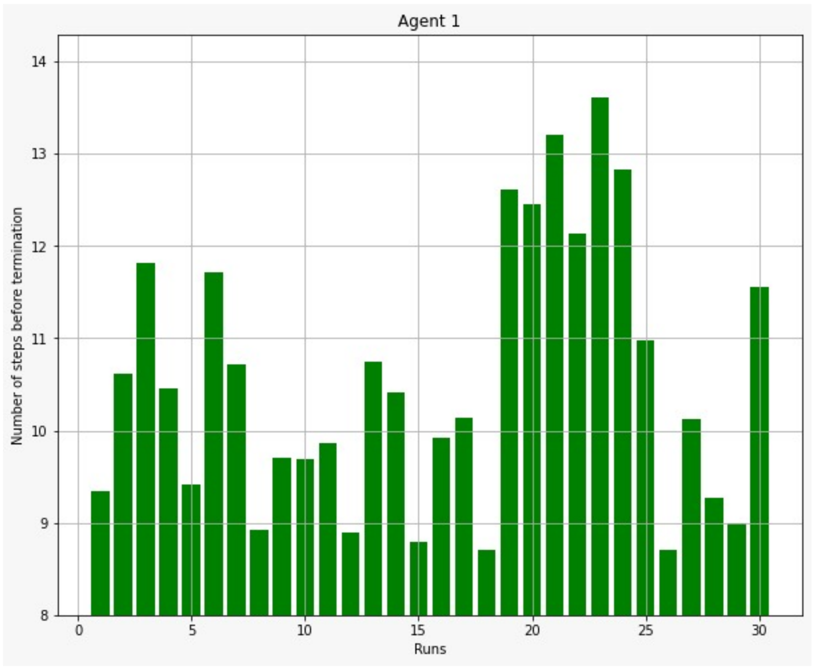


Figure 3: Agent 1: Average number of steps in each run

4 Agent 2

4.1 Drawbacks of Agent 1

Though we have all the information about the prey and the predator, the agent tries to move to the cell in which the prey is currently located. However, the agent doesn't consider the future decisions the prey might take and therefore, has a scope for improvement in this aspect.

Secondly, whenever the distance between the agent and the predator reduces, there is no way to make up for this distance because the predator is always moving optimally toward the agent.

4.2 Agent 2

The main idea of Agent 2 is to address the above-mentioned drawbacks and move in accordance with the future belief states of the prey. We have used the following steps to build the future belief array. In simple words, we consider the next positions in which the prey can be and move in accordance with that.

- Since we know the current position where the prey is, we can build a belief array for the current timestep. This belief array would have the value "1" at the current prey's position and "0" at all other nodes.
- For simplicity, let's imagine we have a 16-node graph and the prey is currently located at the node 9. Therefore, the belief array for this timestep would be as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
								1							

Figure 4: Belief Array at the Current Timestep

- At every timestep, the prey has an equal probability of moving to one of its neighbours including staying in the same cell. Therefore, at timestep t+1, the prey has an equal probability of staying at the current cell and one of its neighbours. Let us imagine that the node 9 has two neighbours, node 8, and 10. Then the belief array at the next timestep would translate to:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
							0.33	0.33	0.33						

Figure 5: Belief Array at the next timestep

- The further we try to look into the future, the more distributed the belief array would be. Therefore, for this agent, we are considering the future belief array after 1 timestep. We then use a custom heuristic that is used on this future belief array.
- The heuristic is the weighted average value for every neighbour to all the positions at which the prey is located. The main idea of the heuristic is to go closer to all these nodes. The heuristic also considers the predator's distance ensuring that we move away from the predator at the same time.

4.3 More on Heuristic

For every agent's neighbour, we go to all the possible nodes where prey might be present and calculate the heuristic. Among all the agent's neighbours including the current cell, the agent decides to move to the node that has the least heuristic value. To calculate the heuristic, we took the probability of the prey being in that node, the distance to that node and the distance of neighbour with the predator into consideration. The following is the code snippet that we used to calculate the heuristic:

```python
def calculateHeuristic(
    self, agentPos, preyPos, predPos, nextPreyPositions, dist, beliefArray, graph
):
    agentNeighbours = Utility().getNeighbours(graph, agentPos)

    heuristics = {}
    for n in agentNeighbours:

        currheuristic = 0
        for i in nextPreyPositions:

            neighbourPredDsitance = dist[n][predPos] + 1

            deno = (neighbourPredDsitance + 0.1) ** 10

            currheuristic += (
                dist[n][i] * (1 - beliefArray[i])
            ) / neighbourPredDsitance

        heuristics[n] = currheuristic

    return heuristics
```

Therefore, the heuristic always tries to move closer to all the nodes in which the prey might exist. Since it is a weighted average, it moves closer to the node that has a higher probability of prey being there than the node that has a lower probability. This calculation is not applicable in Agent 2 and works better for Agent 4. It also takes the predator distance in the denominator which ensures that it is maximized as much as possible.

4.4 Performance and Survivability

By using this future belief state and moving closer to the nodes in which we might find the prey in the future, we were able to outperform the Agent 1's performance considerably. Similar to agent 1, we ran our agent for 30 new graphs and 100 times each in a graph. Our success rate for Agent 2 was in the range [95% - 97%]. We have also observed that the average number of steps it took to catch the prey or get caught by the predator was around 14.

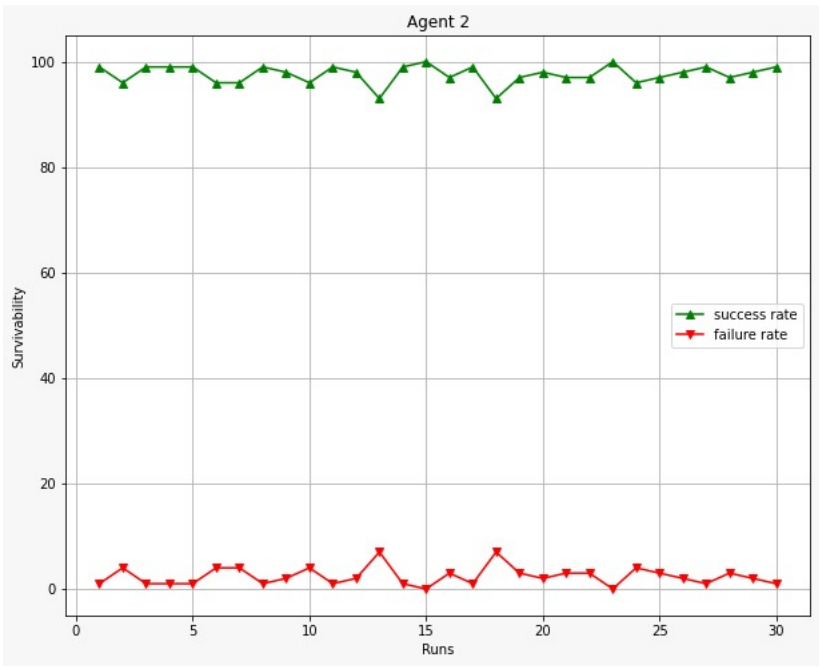


Figure 6: Agent 2: Success Rate vs Failure Rate for 30 runs

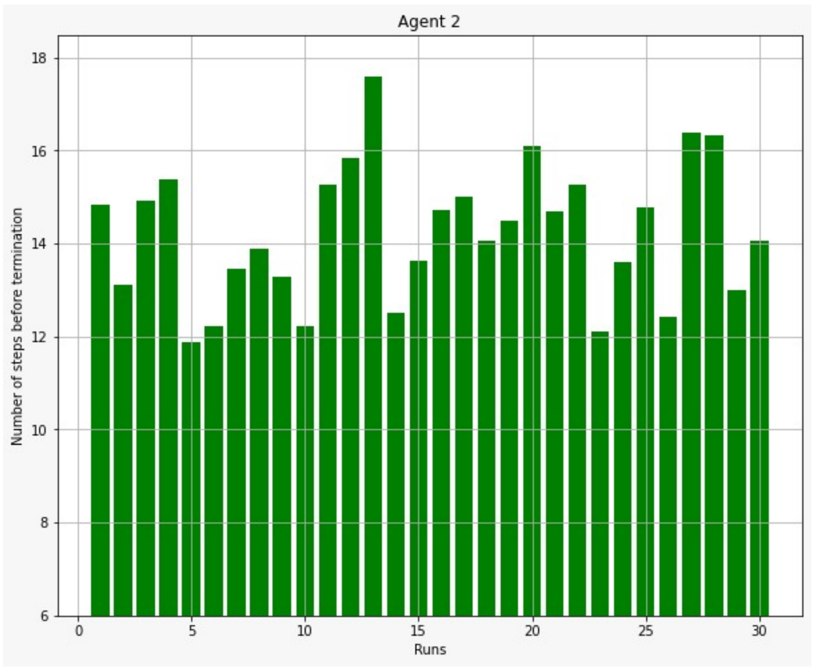


Figure 7: Agent 2: Average number of steps in each run

4.5 Agent 1 vs Agent 2 Comparison

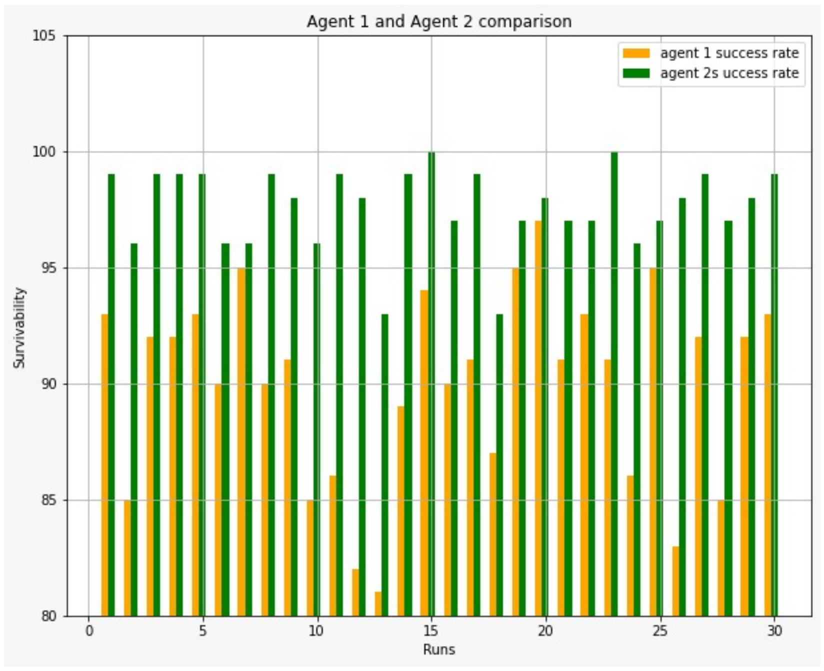


Figure 8: Agent 1 vs Agent 2 Success Rate

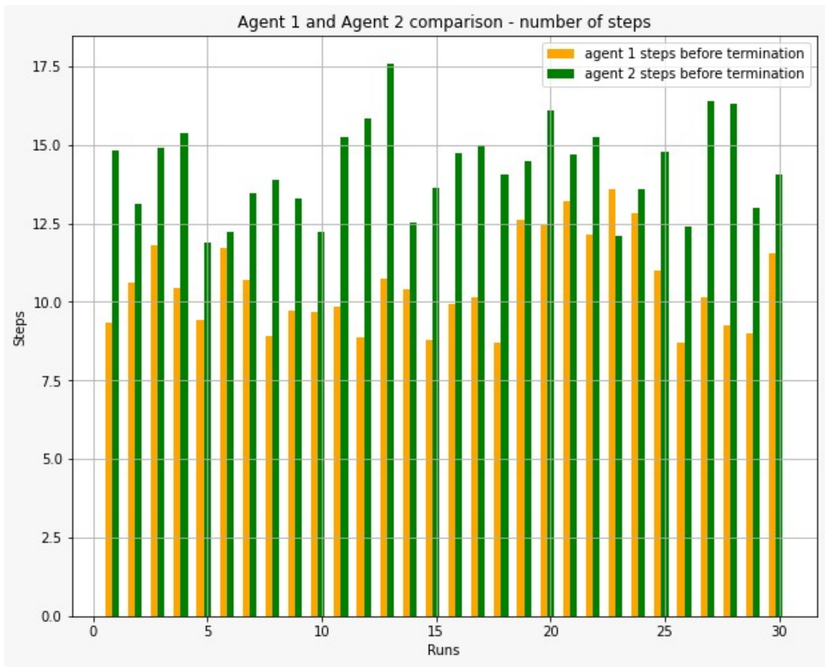


Figure 9: Agent 1 vs Agent 2 Number Of Steps Before Termination

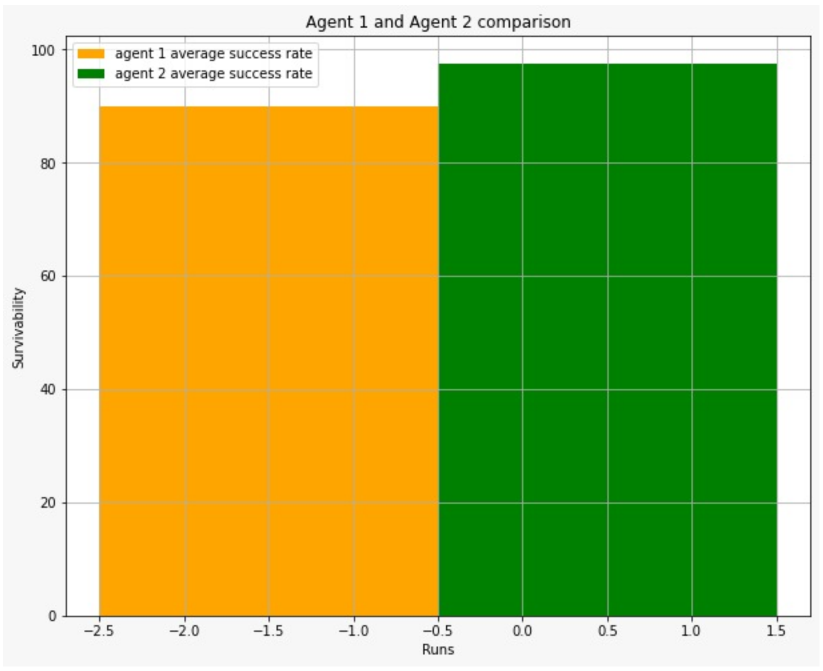


Figure 10: Agent 1 vs Agent 2 Average Success Rate

5 Agent 3

In this environment, the agent always knows the location of the predator but doesn't know the location of the prey. Initially, the agent starts with the information that the prey is not located in the current cell. Now, the agent can scout a node to check if the prey is present in that node. If it is present, then it moves towards that node, if not, it moves towards the node that has the highest probability of containing the node. In this setting, the prey's movement is random and the predator's movement is optimum, it always tries to move towards the agent in the shortest path.

To implement this agent, we use the belief array which denotes the probability of the prey being in that specific node. Whenever an agent moves to a node, it updates its belief at the current position to 0 if the prey is not present. It now gets a chance to scout a node before taking a step. When the agent finds the prey in the scouted node, it updates its belief in that node to be 1 and all the other beliefs become 0. If the prey isn't found, then it updates its beliefs and travels towards the node that has the highest belief. The detailed implementation is explained below.

5.1 Implementation

- Initially, since we don't know the current prey's position, we assume that all the nodes have an equal probability of prey being in that position. However, since we know that the prey is not in the current agent's node (the game ends if the prey is in the agent's node), we can clearly conclude that the probability of prey in the agent's node is 0.
- Therefore, the initial probabilities of all the nodes are **1/49**.
- Now, whenever we scout the node, we will check if the prey is present in that node or not. If the prey is present in that node, then we update the belief array to be 1 at that particular node and 0's at all other nodes because we are certain that the prey is present in that node.
- However, if we fail to find the prey in the scouted node, then we calculate the probability of prey being in the node given the prey is not present in the agent's node and scouted node.

- Let's say we have nodes A - Z, and the node we scouted is D. Let's imagine that there is no prey in node D, then for every node apart from the agent's node, we calculate the following probability:
P(Prey in A — Prey not in D)
= P(Prey in A and not in D) / P(Prey not in D)
= P(Prey in A) * P(Prey not in D — Prey in A) / P(Prey not in D)
= P(Prey in A) * 1 / 1 - P(Prey in D)

- To summarize, before we make the scouted node's probability 0, we multiply every node's probability with:
1 / (1 - P(Prey in the scouted node))

- Now, using the belief array, we assume that the prey is at the node that has the highest probability and try moving towards it.

- Finally, after considering the node with the highest probability, the prey also moves. This movement in the belief array is computed by propagating the probabilities. Since the prey's movement is random among all its neighbours with equal probability, we distribute the probability of each cell to itself and its neighbours equally.

- For example, lets say we are at node A and we wish to distribute the probability of A, assuming B and C are the neighbours of A:
P(Prey in A next) = P(Prey in A next AND Prey in A now) + P(Prey in A next AND Prey in B now) + P(Prey in A next AND Prey in C now) + ...
= [P(Prey in A now) * P(Prey in A next — prey in A now)] + [P(Prey in B now) * P(Prey in A next — prey in B now)] + [P(Prey in C now) * P(Prey in A next — prey in C now)] + ...
= P(A) * 1/degree(A) + P(B) * 1/degree(B) + P(C) * 1/degree(C)

- The above probability is calculated for all the nodes and finally, we get the belief array for the next time step. We again start the same process in the next time step.

5.2 Performance and Survivability

The success rate of Agent 3 primarily depends on how well we are able to predict the prey and how long we are able to survive the wrath of predator. Based on our multiple iterations, the success rate for agent 3 has been 85 % where in each iteration, we ran the agent for 30 new graphs and 100 times in each graph.

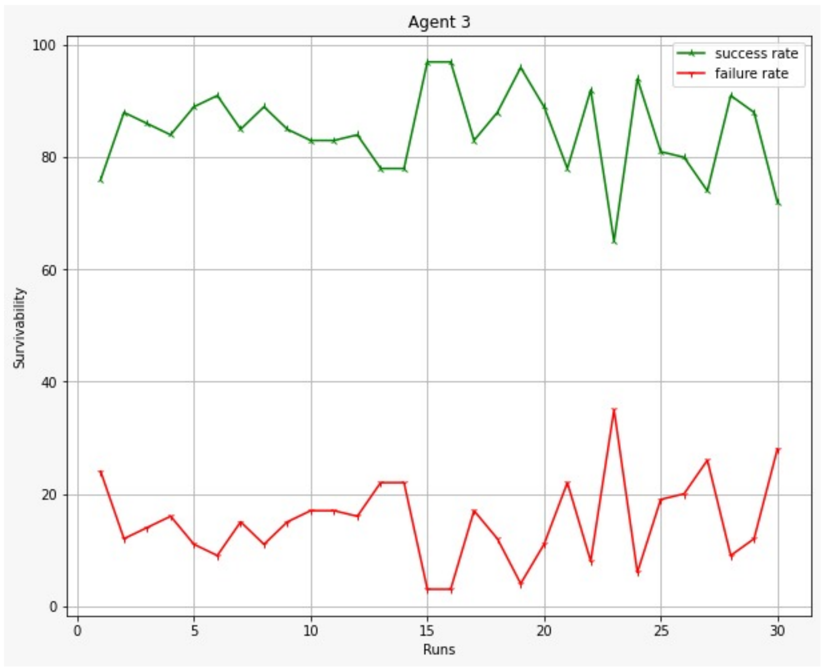


Figure 11: Agent 3 Success Rate and Failure Rate

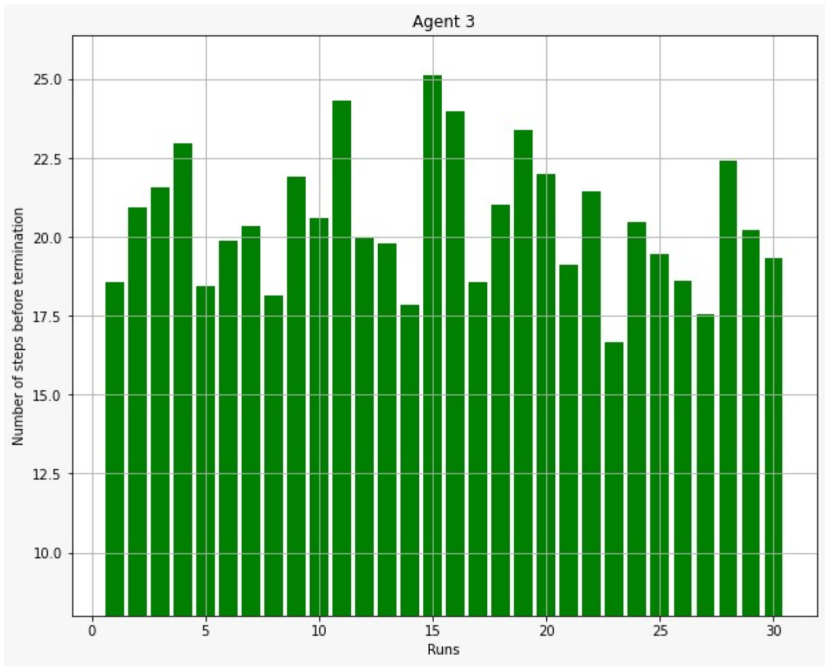


Figure 12: Agent 3 Average Number of Steps Before Termination

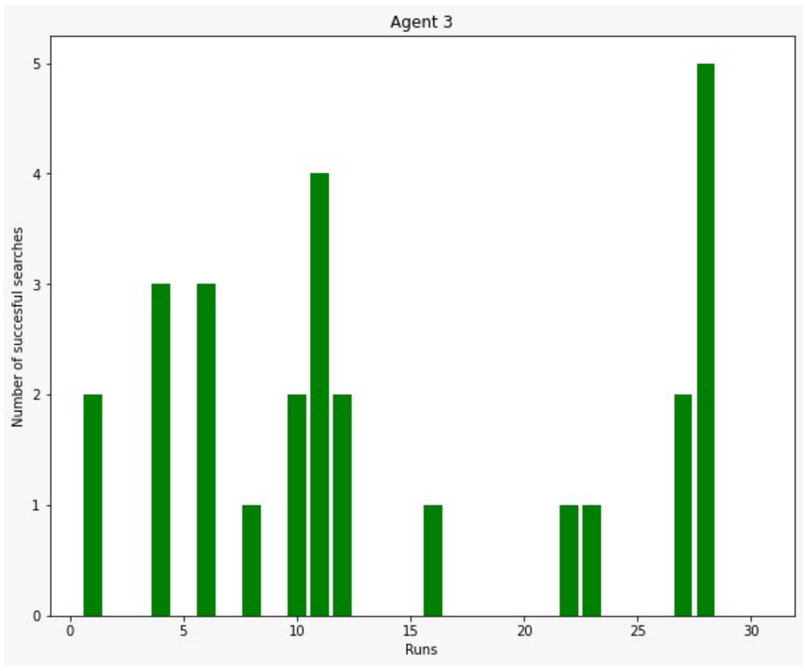


Figure 13: Agent 3 Number of Successful Scouts in Each Iteration

6 Agent 4

This agent has the same setting as that of Agent 3 and we wanted to use the same functionality as our Agent 2. As explained earlier, agent 2 takes a future belief array of the prey and we try to go closer to all the nodes in which the prey might be in the future. We used the same heuristic on our agent's all neighbours including the same node, essentially moving closer to the prey while trying to move away from the predator. In simpler terms, we consider the next steps in which the prey can be and we move in accordance with that.

6.1 Implementation

- At every time step, we have the prey's belief array that explains the probability of the prey being in a particular node.
- We then get an intermediate belief array after scouting a node and translating this information into our belief array.
- Using this belief array, we try to compute the belief array at the next timestep by propagating the values to its neighbours. Since the prey moves with equal probability, we propagate the values equally to itself and its neighbours. (Just like agent 2)
- Let's call this belief array to be the next timestep belief array. We then use this belief array on our heuristic that has been previously explained(from the second agent) to decide on the next step that the agent needs to take.
- The heuristic essentially ensures that the agent moves closer to the prey while moving away from the predator.

6.2 Performance and Survivability

Agent 4 outperforms Agent 3 by a margin. We run the agent for 30 new graphs and 100 times each on a graph and we get a success rate of 94%. The steps, however, increase drastically to around 22 steps thereby also indicating better survivability of the agent.

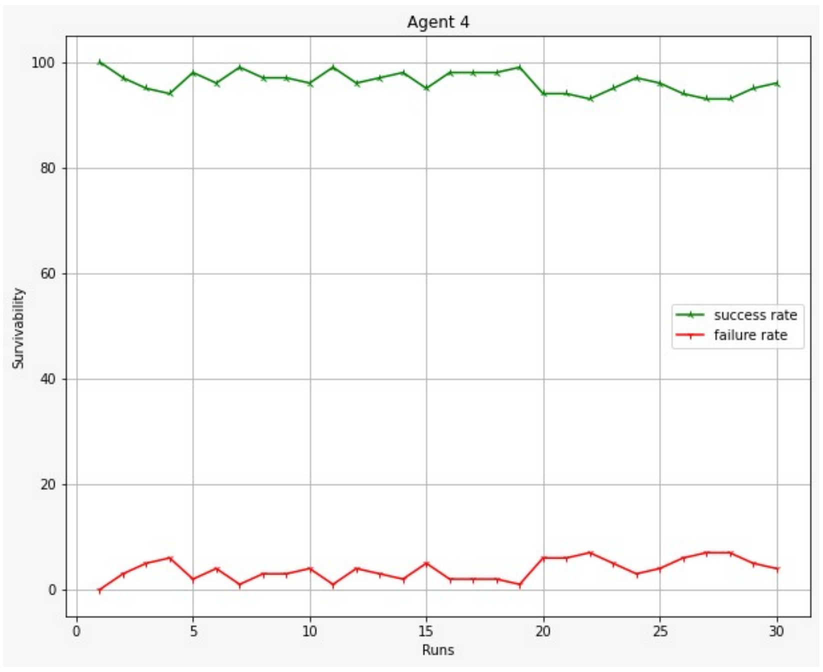


Figure 14: Agent 4 Success Rate and Failure Rate

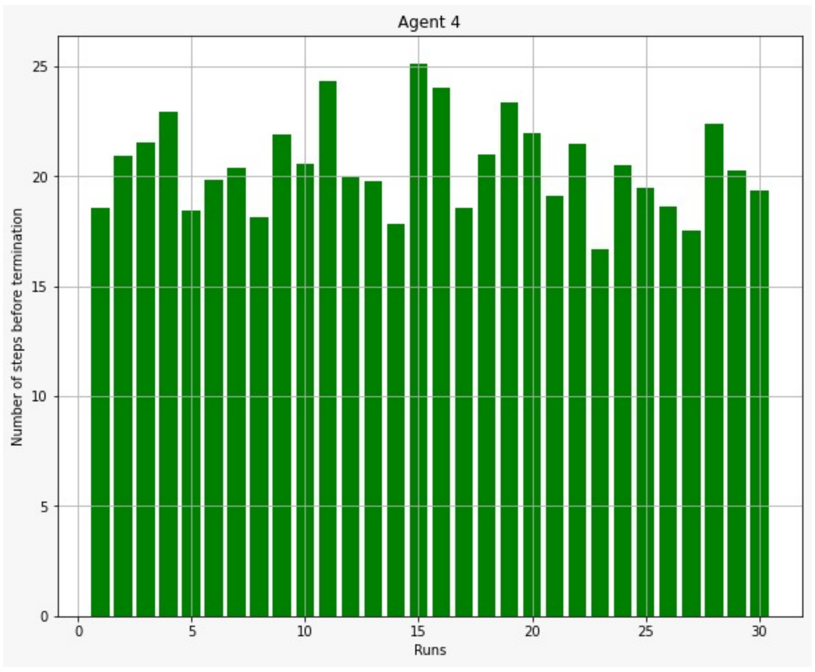


Figure 15: Agent 4 Average number of steps before termination

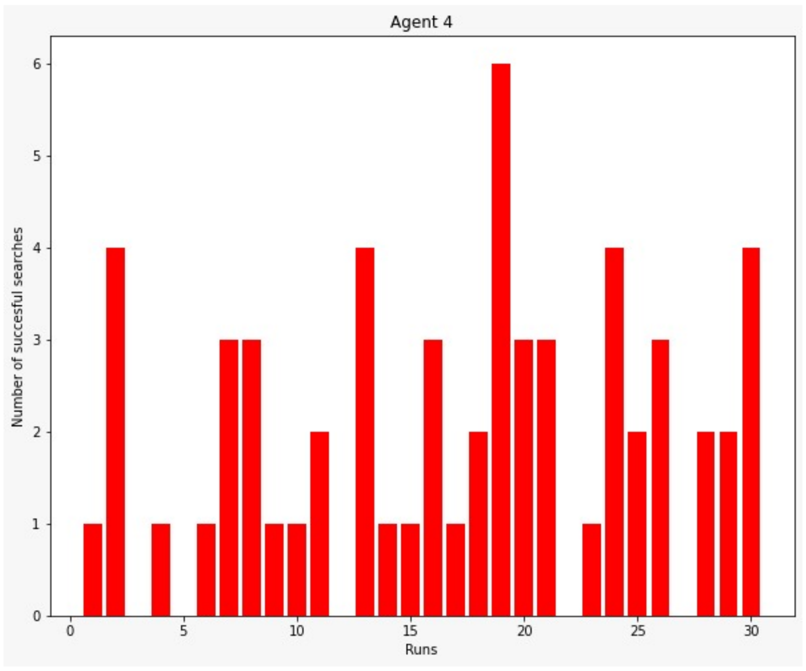


Figure 16: Agent 4 Number of Successful Scouts in each iteration

6.3 Agent 3 vs Agent 4 Comparison

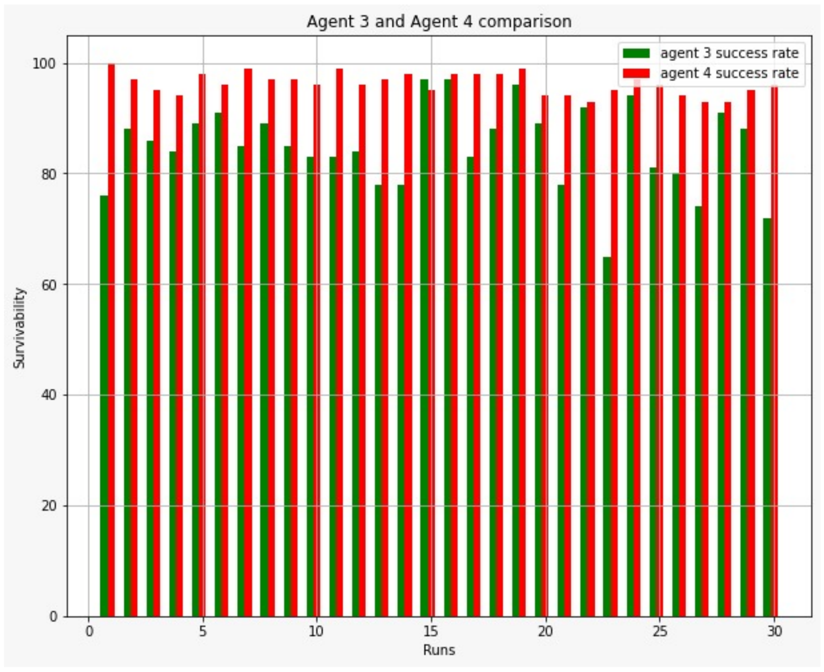


Figure 17: Agent 3 vs Agent 4 Success Rate

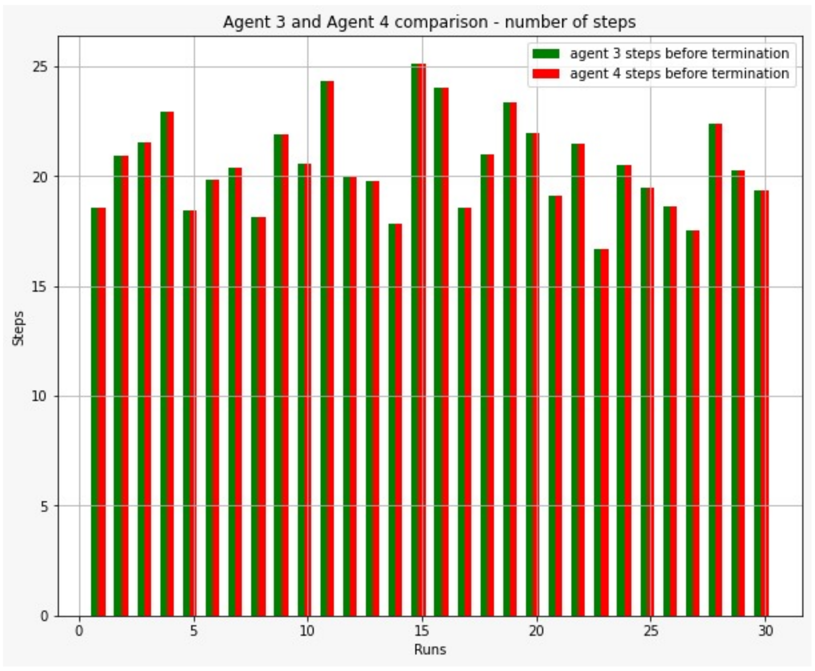


Figure 18: Agent 3 vs Agent 4 Number Of Steps before Termination

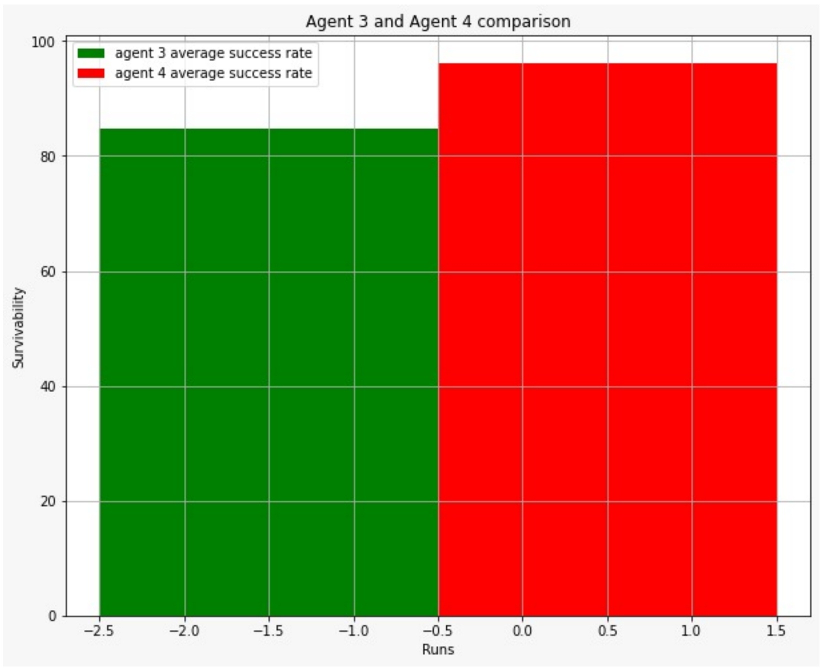


Figure 19: Agent 3 vs Agent 4 Average Success Rate

7 Agent 5

In this setting, the agent always knows the prey's exact location but doesn't know the predators location. It scouts the node with the highest probability and updates its belief array based on that. It then assumes that the predator is located in the node with the highest probability and tries to move away from it. This agent moves in accordance with the agent 1. The agent starts by knowing the initial location of the predator.

7.1 Distracted Predator

Till now, the predator has always moved optimally towards the agent. In this setting, the predator moves probabilistically. It moves towards the agent in shortest path with 0.6 probability and with 0.4 probability, it moves randomly.

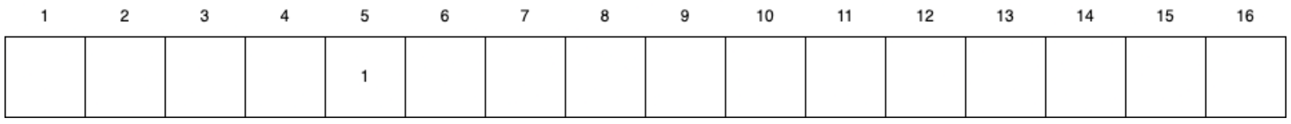
7.2 Implementation

- Initially, since we know the position of the predator, our belief array would have 1 at the predators positions and 0's at all other positions.
- Now, after making the move, we no longer know the location of the predator. Therefore, we update the belief array for the next time step and scout the node with the highest probability of predator being in it.
- If we find the predator in the scouted node, then we are again certain about its location and move in accordance with agent 1. If we dont find the predator in the scouted node, then we make the probability of the predator being in that node 0 and update the belief array, thereby assuming that the predator is in the node with highest probability.

7.3 Predator Probability Updates

The following steps explain a step-wise probability updates for the predator. The same probability update function has been used throughout our further implementation.

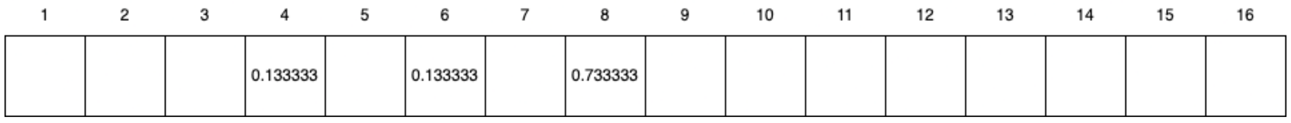
- Initially, since we know the predator's location, we formulate a belief array for predator with 1 in its position and 0 in all other positions.
- We then scout a node with the highest probability. If the predator exists in that node, then we mark the probability of that node to be 1.
- If the predator doesn't exist in that node, we then update our belief array. For every node except the scouted node, we change the value as:
  P(A) = P(A) / (1 - P(S)) where A is every node except the scouted node and S is the scouted node. *As derived in Agent 3*
- Now, in this updated belief array, we assume that the predator is in the node with the highest probability and the agent moves closer to the prey while moving away from the predator, like in Agent 1.
- Finally, since the predator is not optimum, we need to perculate the probability and calculate the next time step's belief array since the predator takes a move.
- To calculate this belief array, we take the existing probability in the belief array and assign 0.6 of it to the node in the shortest path towards the agent. We then assign 0.4 of the value to all the neighbours of the predator.
- For example, let's assume we thought the predator was at 5(the nodes connected to it are 4,6,8) and upon scouting we understood that it is there. Assume the intelligent move(closest to agent) was towards 8. So node 8 is given 0.6p of what was there in 5 =0.6 and rest of the 0.4p is given to 3 of the neighbours as 0.4p/(degree of 5=3)=0.133333333, so the probabilities in the next time step would be



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
				1											

Figure 20: Agent 5 Before Belief Update

- After the belief Update



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			0.133333		0.133333		0.733333								

Figure 21: Agent 5 After Belief Update

- Therefore, as derived in Agent 3, we traverse through all the nodes and assign these probabilities and use the new belief array at the next time step.

7.4 Performance and Survivability

On comparison with the Agent 3, we found that the number of successful surveys in agent 5 is more. This is primarily because the probability distribution of the predator is not as distributed as that of the prey. Though the predator moves probabilistically, it moves in the shortest path with 0.6 probability and the scout has more chance to find the right node the predator is in. The success rate of agent 5 is 81% with an average of 14 steps to close the chase.

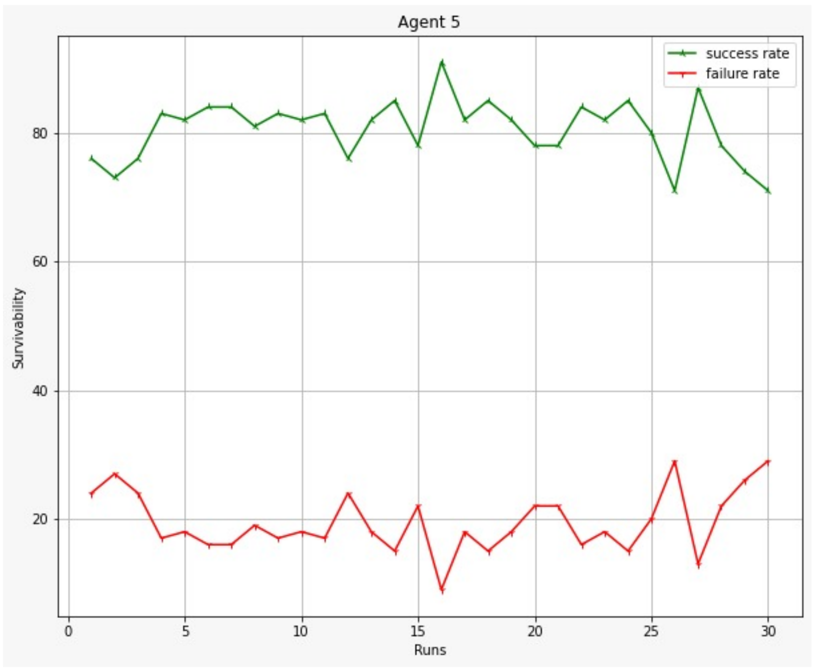


Figure 22: Agent 5 Success vs Failure

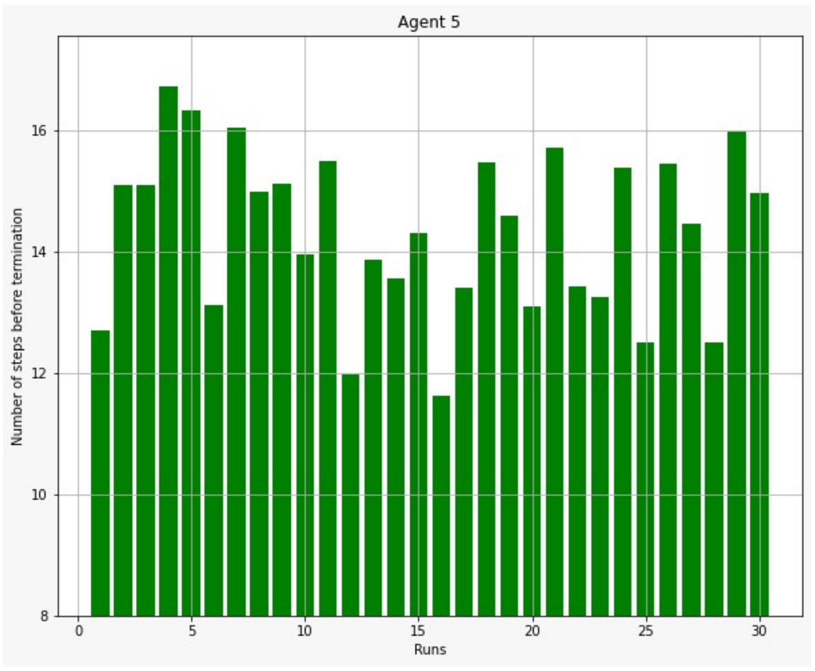


Figure 23: Agent 5 average number steps in each run

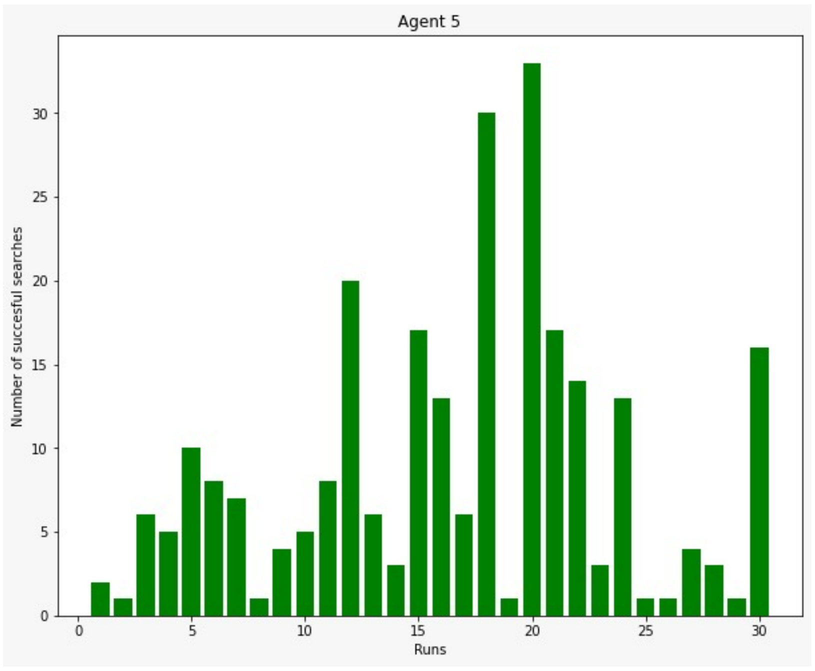


Figure 24: Agent 5 Number of successful scouts in each run

8 Agent 6

The main idea of Agent 6 is that instead of moving away from the node that has the highest probability of predator being in that node, we essentially move away from all the nodes in which the predator can be in the next timestep. That is, in agent 5, we have considered the predator to be in the node with the highest probability but failed to consider the nodes in which the predator can go to in the next timestep. Therefore, to do this, we have defined a custom heuristic that is run on every agent's neighbour for all the neighbours of the predator node. (The node which has the highest probability in the predator belief array)

8.1 Implementation

- The agent 6 knows the initial location of the predator, so we create a belief array with the value 1 in the predator's position and 0 in all other cells.
- Now, since we know the predator's location, we take the predators neighbours (the nodes in which the predator can be in the next time step) and calculate a heuristic for the agent's every neighbour to all these predators locations.
- We now pick the the maximum value of this heuristic and move to that node.
- Now since we have moved, we move the predator as described in Agent 5 and update the belief array of the predator. (Also described in Agent 5).

8.2 Heuristic

For this agent, we have used a different heuristic as compared to our previous agents. Previously, we had a more distributed probability array and we had to use a heuristic that considers all the nodes. However, in this setup, our belief array is more converged. Therefore, our heuristic for this agent is defined as:

2 * distance[agentNeighbour][i] - distance[Prey][AgentNeighbour] * (1 - beliefArray[i]) where i is defined as all the predator's neighbours and beliefArray is the predator's belief array.

The above heuristic works well in this case because it considers the factors of prey's distance and predator's distance along with the probability of the agent not being in that neighbour. We take the maximum value of this heuristic as thats node is more desirable.

```

def calculateHeuristic(
     self, agentPos, preyPos, nextPredPosition, dist, beliefArray, graph
):
    agentNeighbours = Utility().getNeighbours(graph, agentPos)
    agentNeighbours.append(agentPos)

    predatorNeighbours= Utility().getNeighbours(graph, nextPredPosition)

     heuristics = {}
    for n in agentNeighbours:

        currheuristic = 0
        for i in predatorNeighbours:

            currheuristic += (dist[n][i]*2-dist[preyPos][n]) * (1 - beliefArray[i])

        heuristics[n] = -currheuristic/len(predatorNeighbours)

    return heuristics
```

8.3 Performance and Survivability

Using the above custom heuristic and looking into the future, we were able to outperform Agent 5 by a considerable margin. Our success rate was 87%.

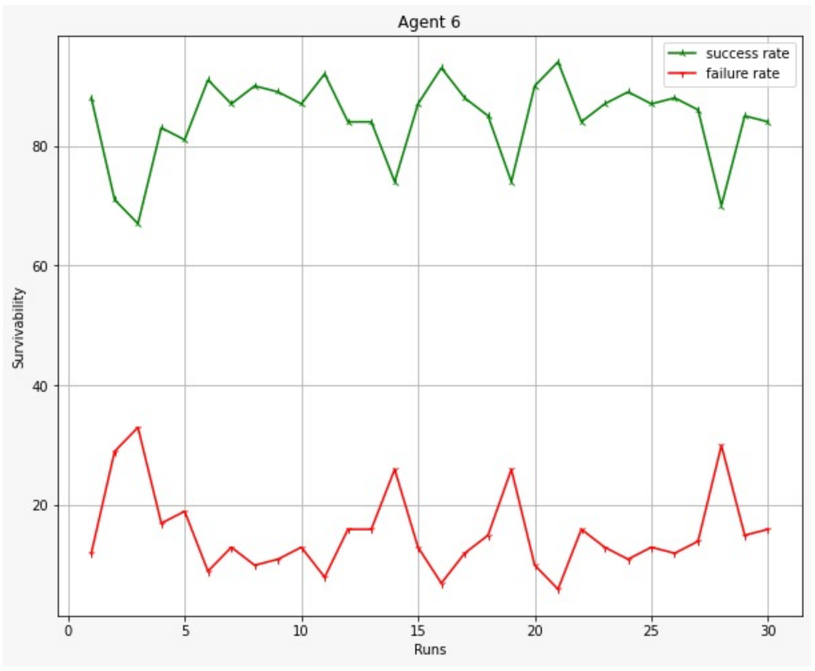


Figure 25: Agent 6 Success vs Failure

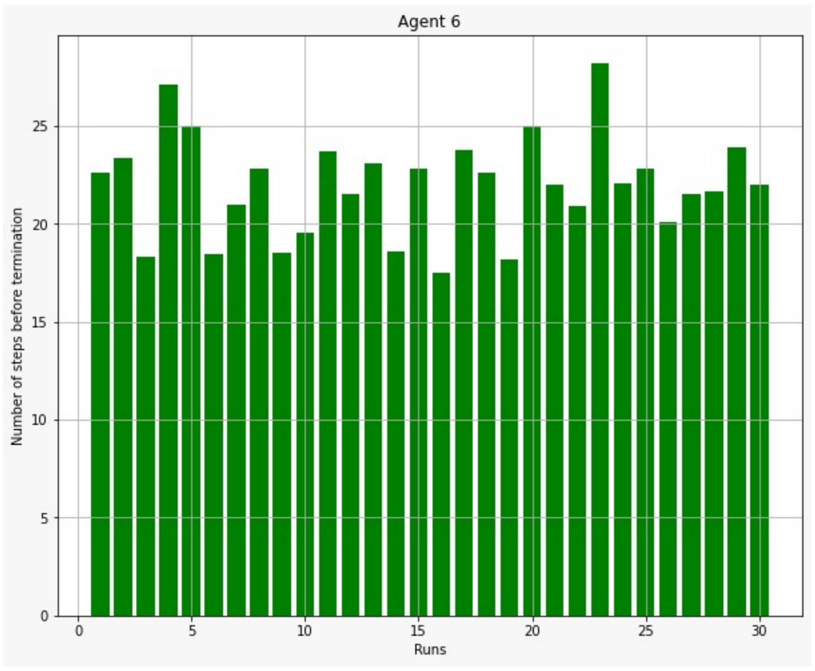


Figure 26: Agent 6 Number Of Steps before Termination

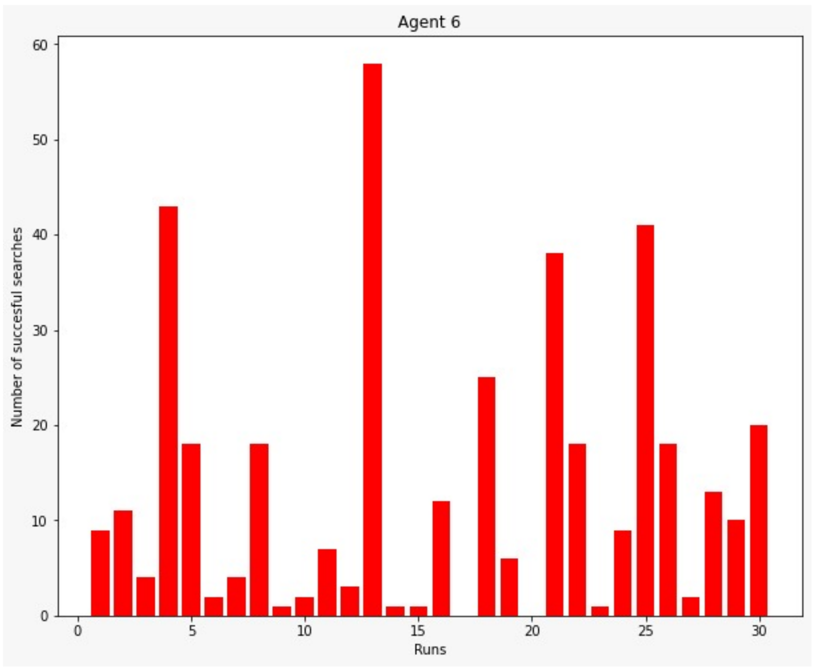


Figure 27: Agent 6 Number of success scouts in each run

8.4 Agent 5 vs Agent 6 Comparison

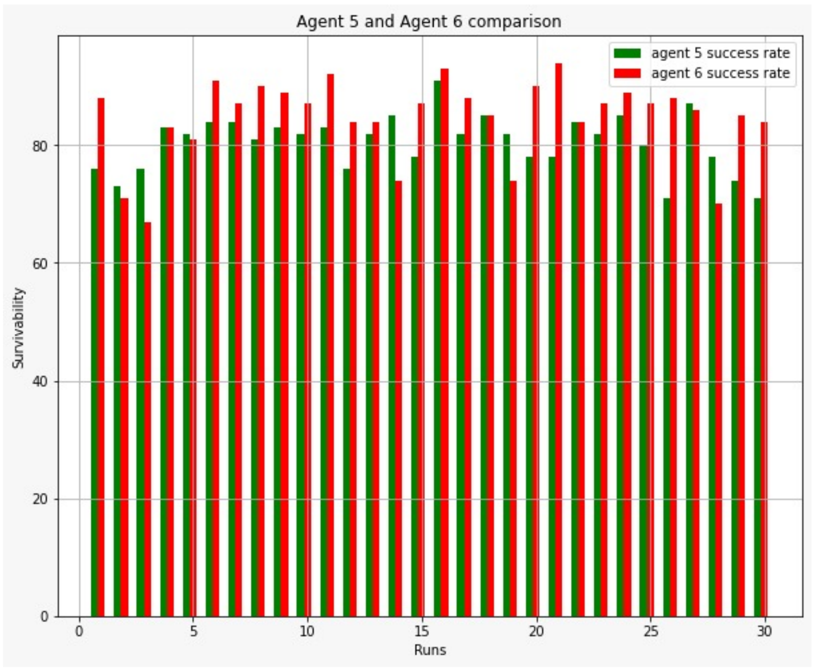


Figure 28: Agent 5 vs Agent 6 Success Rate

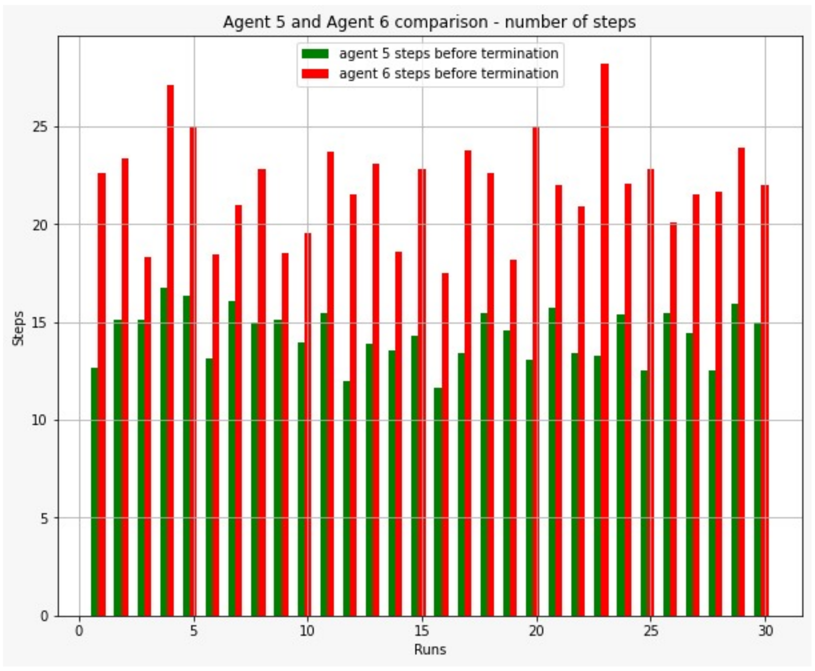


Figure 29: Agent 5 vs Agent 6 Number Of Steps before Termination

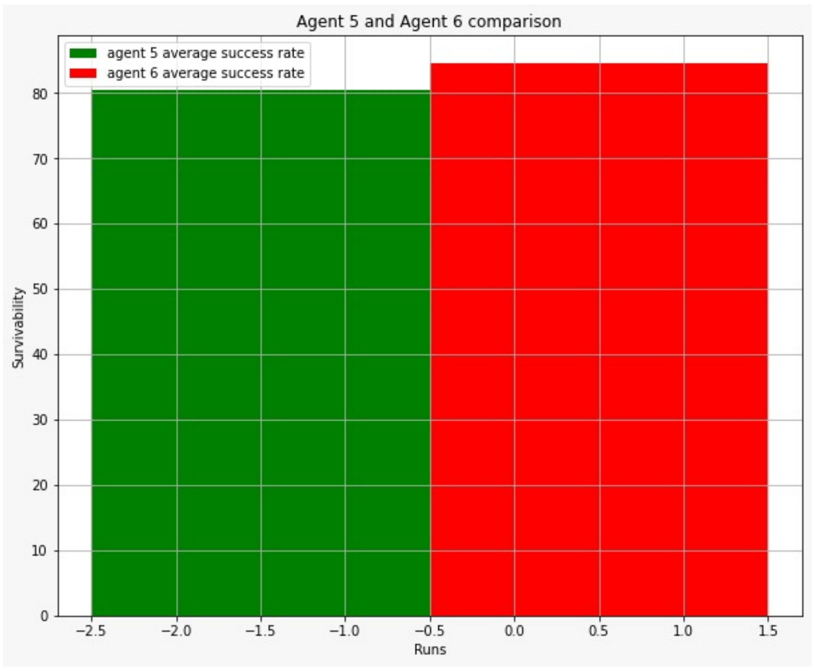


Figure 30: Agent 5 vs Agent 6 Average Success Rate

9 Agent 7

In this combined partial setting, we don't know the locations of both the prey and predator. Therefore, we maintain two belief arrays. One maintains the probability of prey being in that node and the other maintains the probability of the predator being in that node. Our main idea throughout this agent is to move away from the node that has the highest probability of predator being in it and move closer to the node that has the highest probability of prey being in it.

9.1 Implementation

- Initially, we start by knowing the current location of the predator. Therefore, the predator's belief array is created with 1 in the predators position and 0 in all other positions.
- However, we also dont know the initial location of the prey. Therefore, just like agent 3, we start off by having a uniform probability to all the nodes.
- At every time step, we check if we are certain where the predator is. If we are not certain where the predator is, then we scout for predator because we prioritize moving away from the predator.
- However, if we are certain where the predator is, then we scout for the prey and try to move closer to the prey.
- The point that needs to be noted is that whenever we scout a node, it gives out information about both the prey as well as the predator.
- After scouting the node, we update the prey's and predator's probability array depending on the conditions of the scouted node. It can contain the prey but not the predator, or it can contain both or none of them.
- We update the prey's belief array using the conditions of Agent 3 and update the predator's belief array using the conditions of agent 5, both of which have been explained before.
- After updating the probability arrays, we assume that the prey is in the node with the highest probability in the prey's belief array and the predator is in the node with the highest probability in the predator's belief array.
- Finally, after each timestep, we update the predator's and prey's belief array as mentioned in Agent 5 and Agent 3 respectively for the next time step. Since the predator doesn't move optimally, the update function of Agent 5 is used in this setup.

9.2 Performance and Survivability

Using the above conditions, our average success rate for Agent 7 was 75% and the average number of steps before termination was 25.

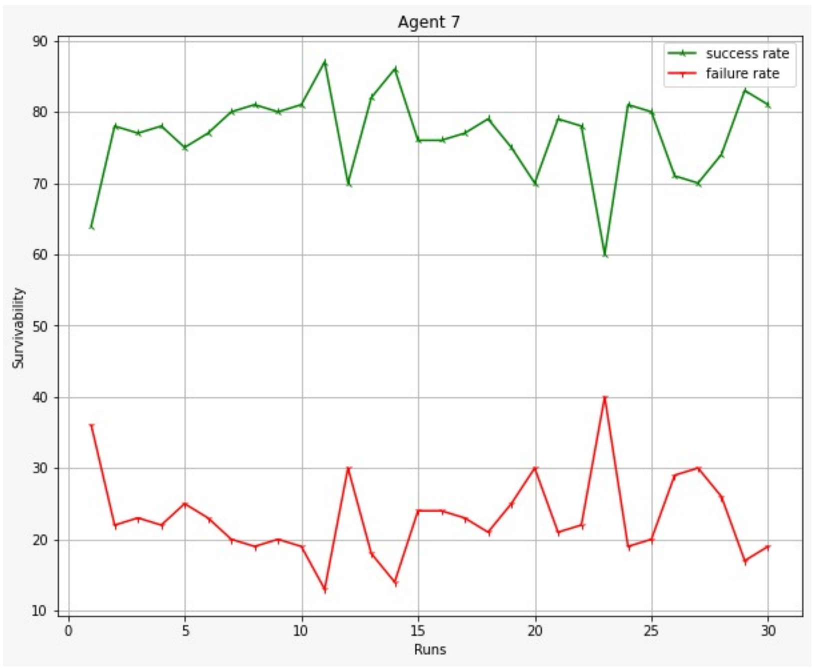


Figure 31: Agent 7 Success vs Failure Rate

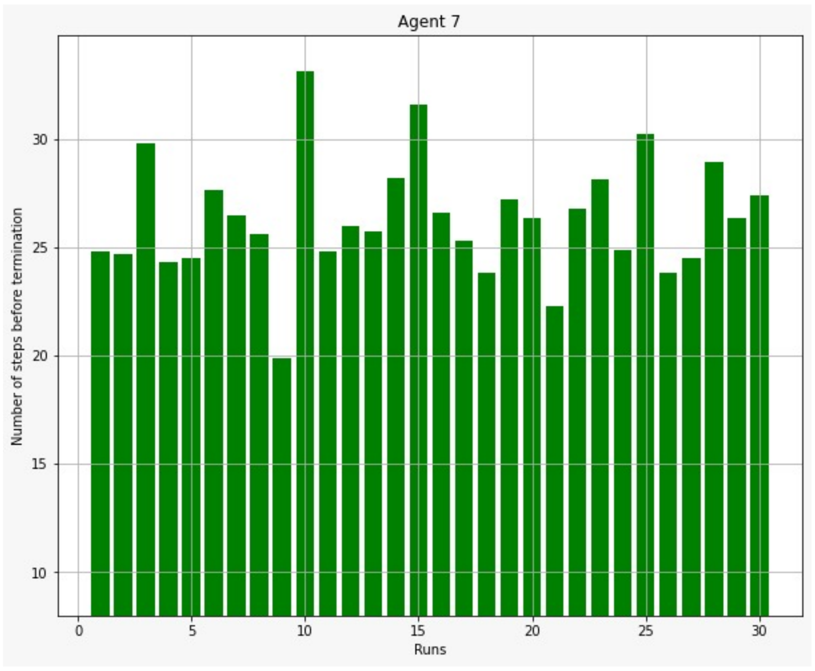


Figure 32: Agent 7 Average number steps before termination in each run

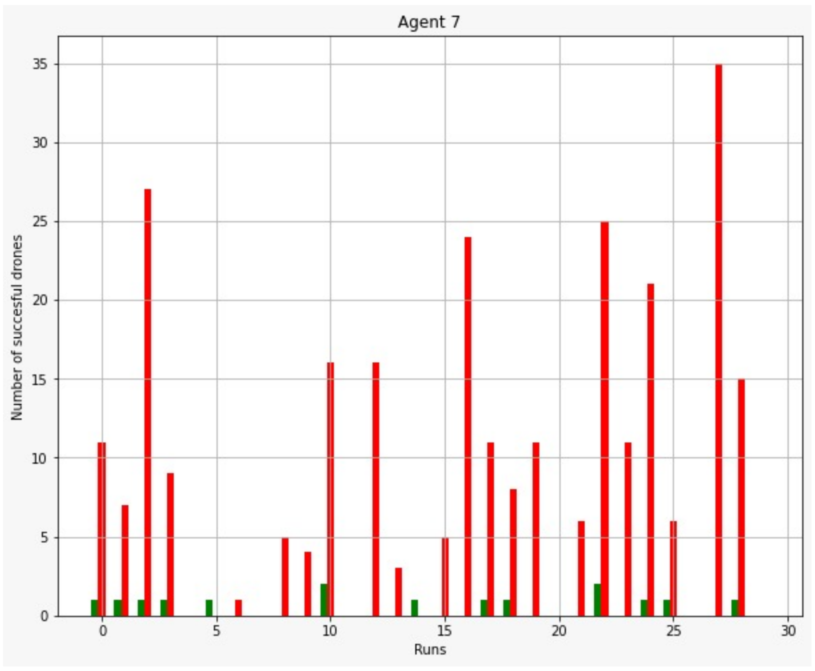


Figure 33: Agent 7 Number of successful scouts in each run

10 Agent 8

Agent 7 considers the current predator's and prey's nodes and tries to optimize the agent's moves based on the current position. However, we were able to improve our agent 7 by considering the next node in which the prey and predator can be and optimizing the agent's movement based on their location.

10.1 Implementation

- Initially, we start by knowing the current location of the predator. Therefore, the predator's belief array is created with 1 in the predators position and 0 in all other positions.
- However, we dont know the initial location of the prey. Therefore, just like agent 3, we start off by having a uniform probability to all the nodes.
- Just like Agent 7, at every time step, we check if we are certain where the predator is. If we are not certain, we scout for the predator, else we scout for node and update our belief arrays.
- After updating the belief arrays, we now consider the next positions in which the prey and predator might go and try to move away from the predator while moving closer to the prey.
- We do this by computing a heuristic, an extension of Agent 6's heuristic.
- In agent 6, we considered the prey's distance to all the predators nodes. As an extension to that heuristic, we have considered the weighted average of all the prey's neighbours distance to the agent's neighbour along with the predator's neighbours distances.
- We have also considered the probability of the prey and predator in those nodes and compute the heuristic.
- We then pick the node that has the maximum value of the heuristic and move to that node.
- After moving, we update the prey's and predator's belief arrays as mentioned in Agent 7.

10.2 Heuristic

For this agent, we have extended the heuristic of Agent 6. In Agent 6, we had a belief array for Predator. Therefore, we extend the same functionality for the prey. The reason this works well in this environment is because though we consider the prey's neighbours(the next node the prey might go to) and the predator's neighbours(the next node the predator might go to), our heuristic is the weighted average of both of these. Since our heuristic takes into consideration the probability of the predator not being in the node and the probability of the prey being in a node, the value that has the maximum value of this is the best choice to take.

```
def calculateHeuristic(
    self, agentPos, nextPreyPosition, nextPredPosition, dist, beliefArrayPred, beliefArrayPrey, graph
):
    agentNeighbours = Utility().getNeighbours(graph, agentPos)
    agentNeighbours.append(agentPos)

    preyNeighbours=Utility().getNeighbours(graph, nextPreyPosition)
    preyNeighbours.append(nextPreyPosition)
    predatorNeighbours= Utility().getNeighbours(graph, nextPredPosition)

    heuristics = {}
    for n in agentNeighbours:

        currheuristic = 0
        for i in predatorNeighbours:

            currheuristic += (dist[n][i]*2-dist[nextPreyPosition][n]) * (1 - beliefArrayPred[i])

        heuristics[n] = -currheuristic/len(predatorNeighbours)
        for i in preyNeighbours:

            currheuristic += (dist[n][i]*1-dist[nextPredPosition][n]*2) * (beliefArrayPrey[i])

        heuristics[n] += currheuristic/len(preyNeighbours)


    return heuristics
```

10.3 Performance and Survivability

Using the above heuristic, we were able to outperform agent 7. On an average, we were able to achieve a success rate of 82% and the average number of steps before termination were around 30.

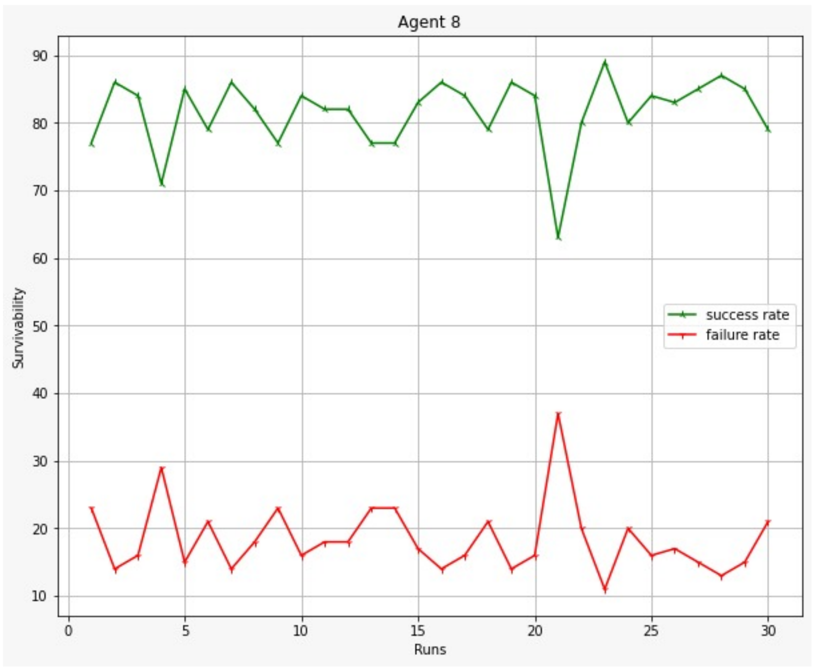


Figure 34: Agent 8 Success vs Failure Rate

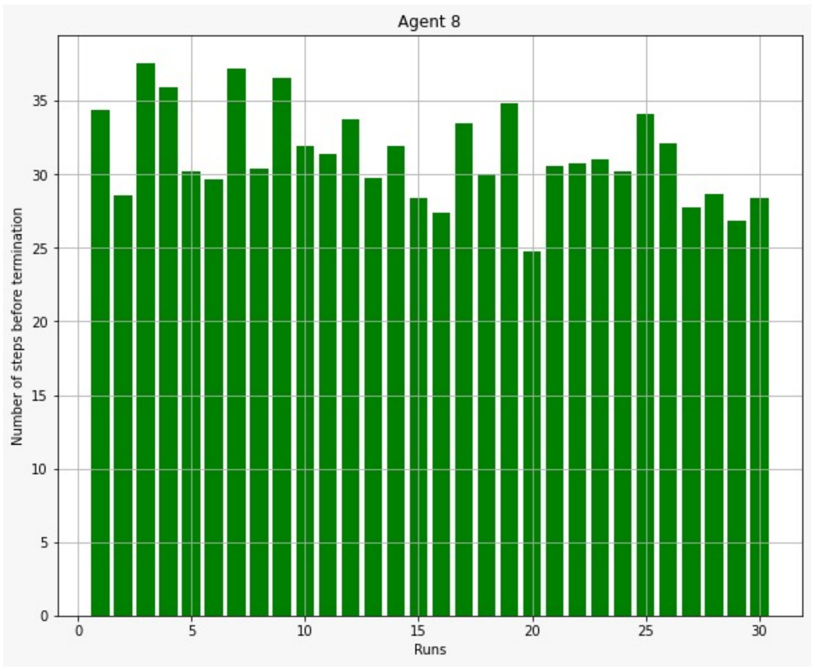


Figure 35: Agent 8 Average number steps in each run

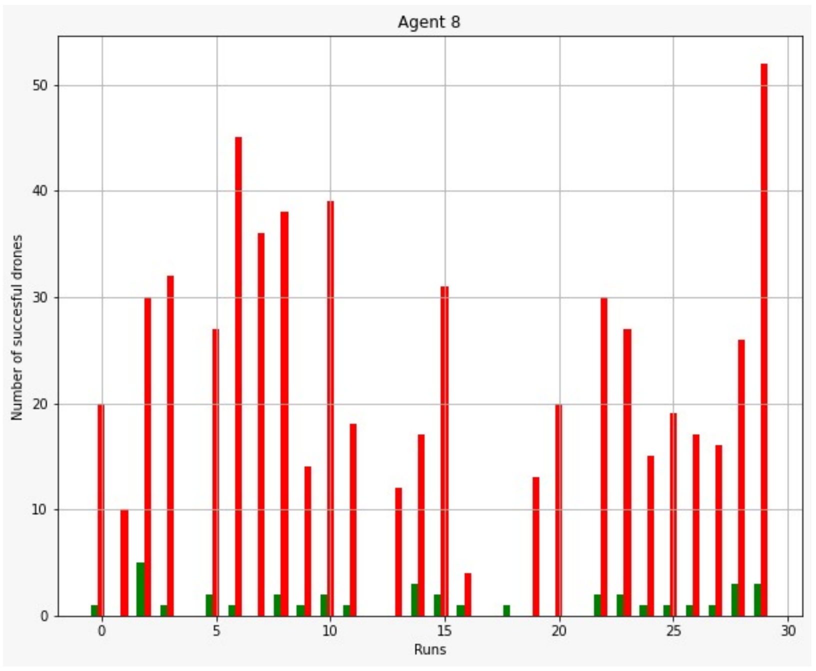


Figure 36: Agent 8 Number of successful scouts in each run

10.4 Agent 7 vs Agent 8 Comparison

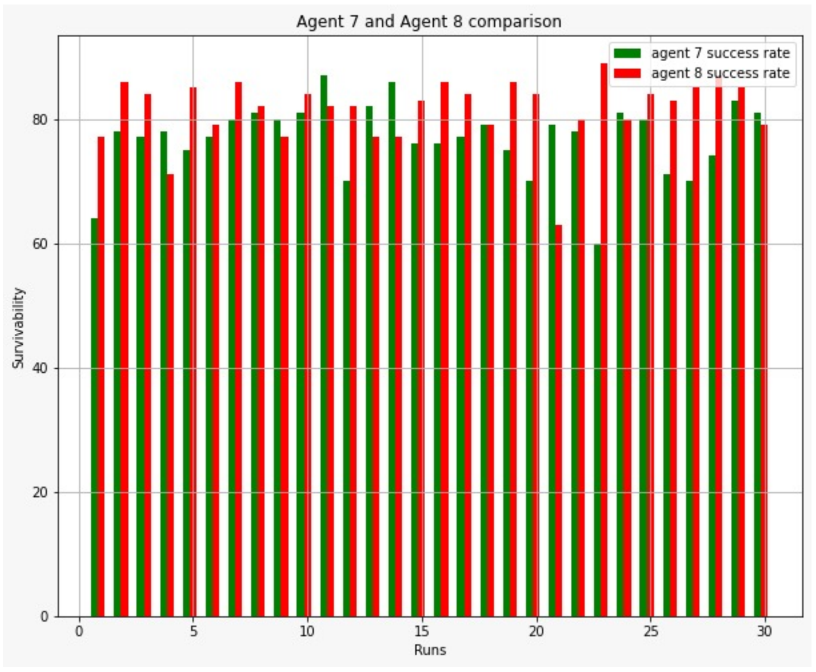


Figure 37: Agent 7 vs Agent 8 Success vs Failure Rate

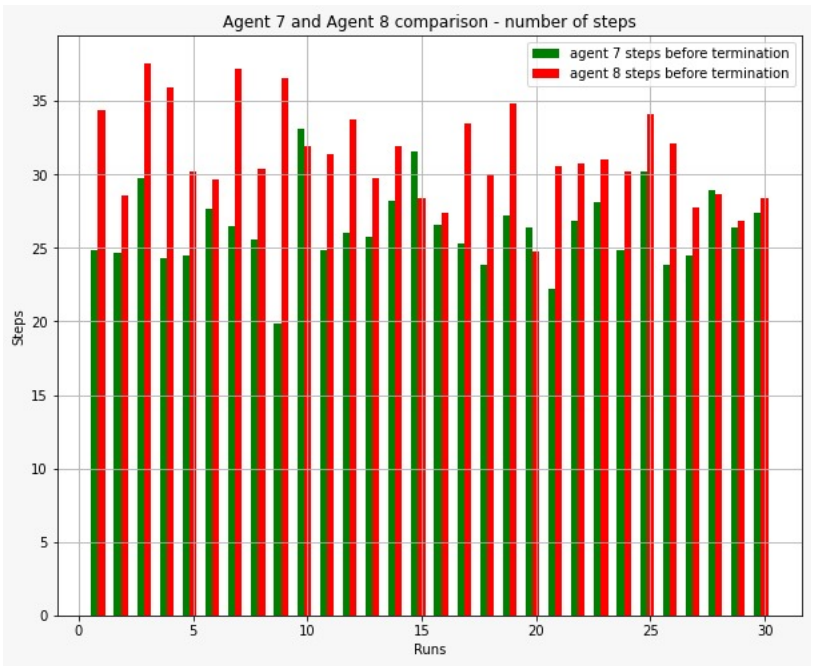


Figure 38: Agent 7 vs Agent 8 Average number steps in each run

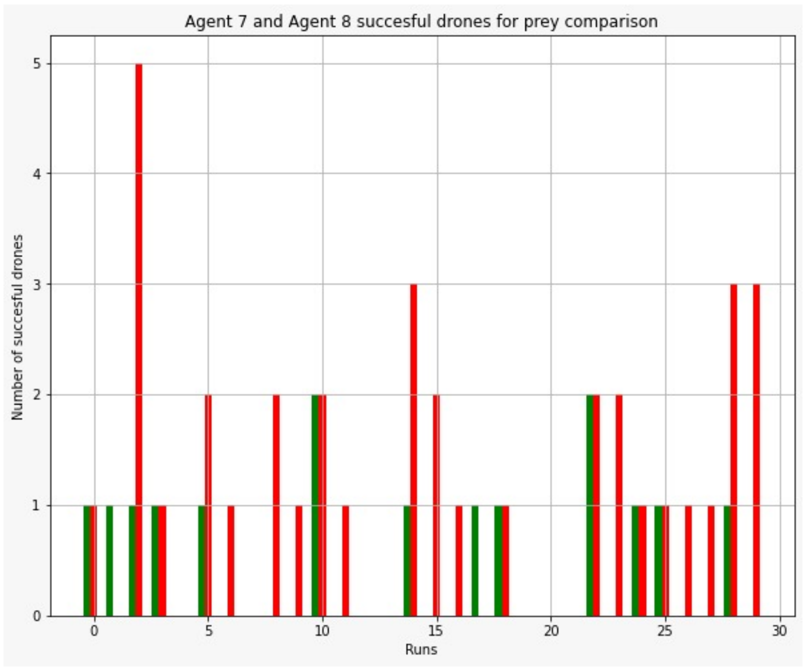


Figure 39: Agent 7 vs Agent 8 Number of successful scouts for prey in each run

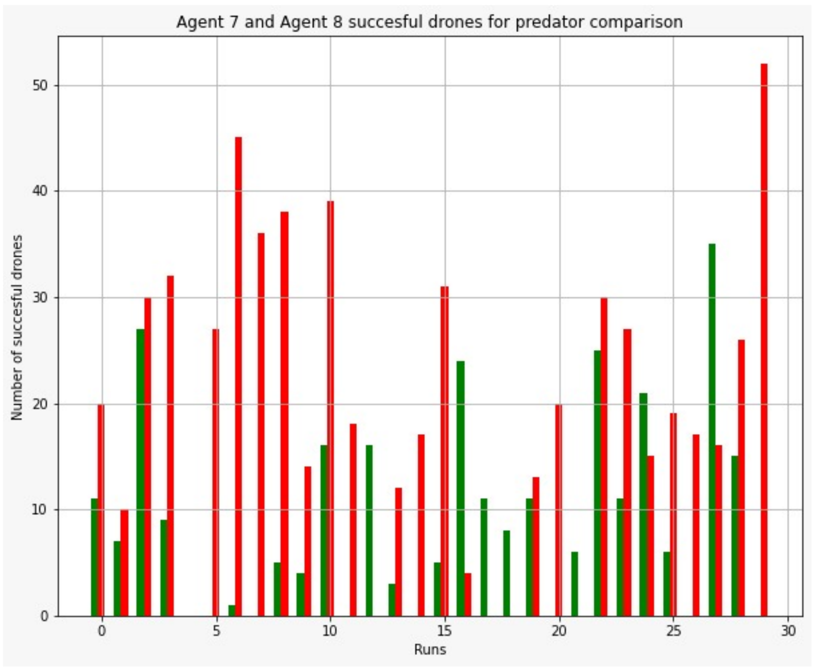


Figure 40: Agent 7 vs Agent 8 Number of successful scouts for predator in each run

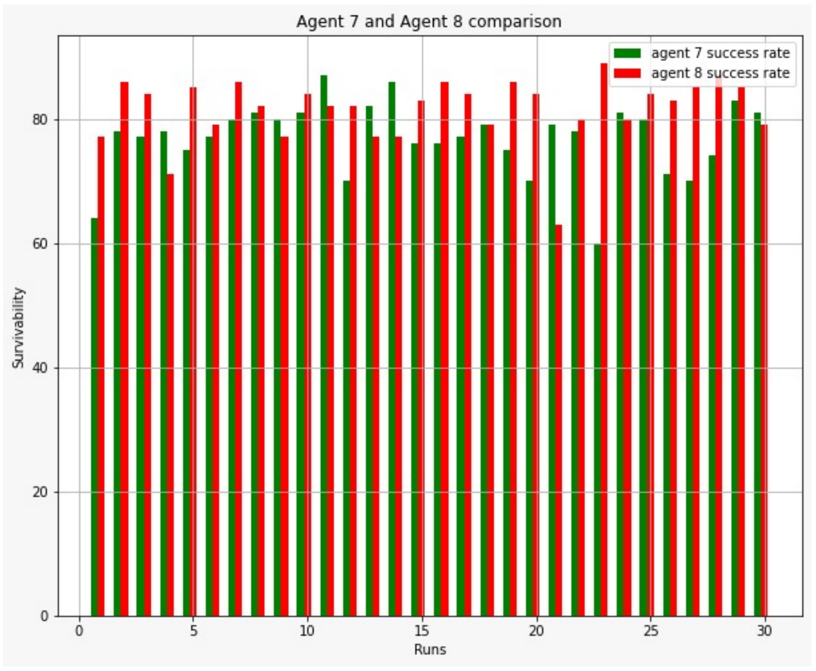


Figure 41: Agent 7 vs Agent 8 Success Rate Comparison

11 Analysis

In our implementation, we have found out that whenever our even agents beat their odd counterparts, it was always because our even agents took the future belief of the prey and predator whenever possible into consideration before making the decision. All our heuristics were designed in a way that would optimize this future belief and move according to the prey and predator's next steps. Ideally, the further we increase this future setting, the more optimum our algorithms should work. However, that's not the case in real life because the more we try to look into the future, the more scattered our belief arrays would become and it would be difficult to track the prey's and predator's future positions exactly.

Therefore, the trend that we observed was that as we increased this future scope, we got somewhat better results but after a point, the success rate reduced drastically.

The second trend that we observed was that the success rate hasn't increased noticeably when we try to increase the time steps of the future. Therefore, for this project, we have taken this timestep to be 1.

The agents are able to use the information from the predator's belief update in a better way rather than that from the prey's belief, as it is more randomly spread. In the case of the Predator belief update, we are initially certain where the predator is and there is more probability that it takes the intelligent move(0.6) rather than a random move.

Also, it is evident that having the information that there is a faulty drone in agent 7, and altering the belief update with it, enabled us to see a considerable increase in the success rate of the agent as compared to not using the information.

We believe that our odd agents are well utilizing the current information and making the decision based on the current information. However, as soon as the agent moves, the prey and predator also move and the current information changes. We tried to address this in our even agents by considering the information of the possible locations of prey and predator in the next time step.

12 Defective Drone Setting

In this environment, the drone that we use to scout is defective. That is, if something is actually occupying a node being surveyed, there is a 0.1 probability that it gets reported as unoccupied. We had to make minimal changes to implement this in our Agent 7 and Agent 8.

Since there is a chance of false negatives (i.e), scout gives us wrong information even if the prey or predator is present in that particular node, we can compensate for this in the belief update by not making the scouted node as zero , instead hold 0.1*p-old in it, as there is 0.1 probability that it was there and the drone reported incorrectly, and since we are doing this. The factor to multiply with each term would change to 1/(1-0.9*P-old[scout]) P-new[scout]=0.1*P-old[scout]

12.1 Defective Agent 7 Performance

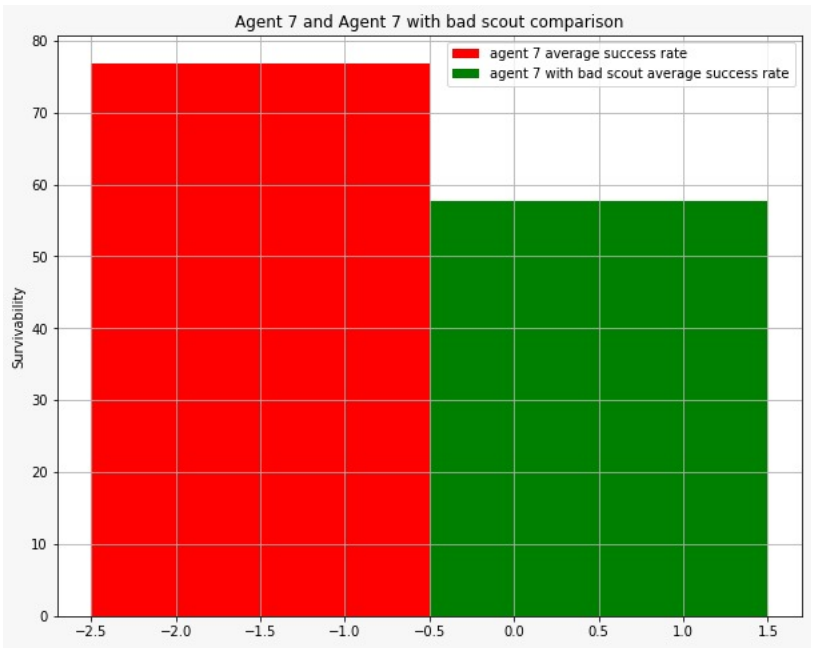


Figure 42: Agent 7 vs Agent 7 with bad scout Success Rate

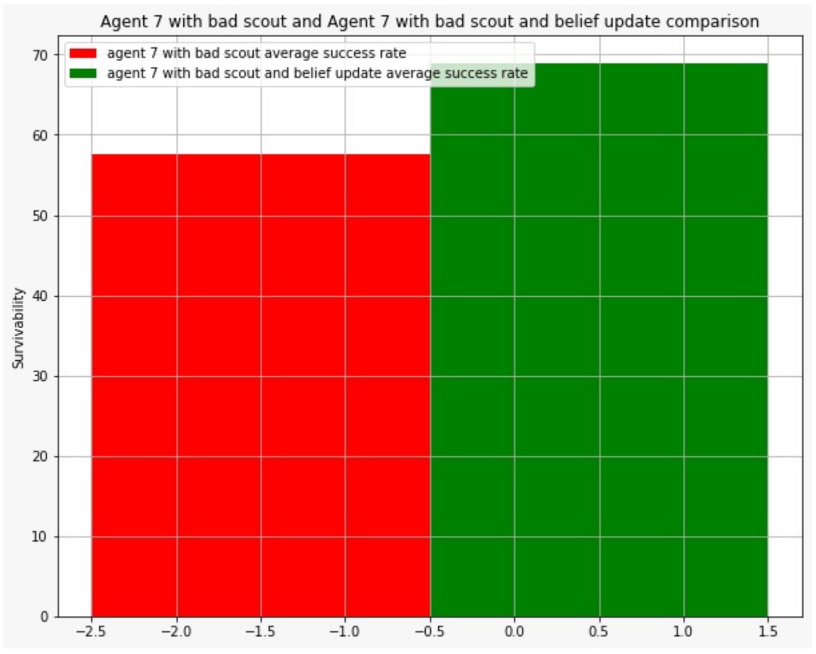


Figure 43: Agent 7 with bad scout vs Agent 7 with bad scout belief update

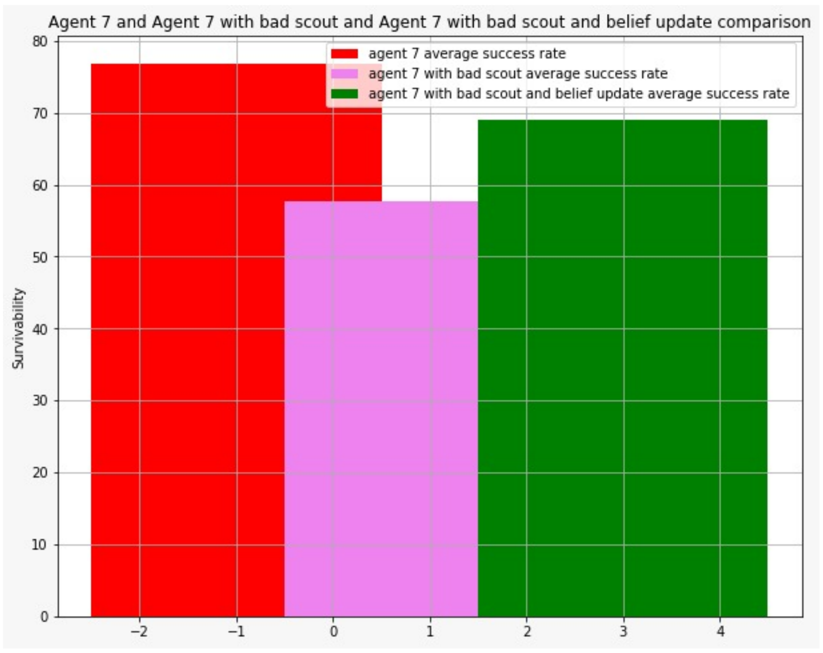


Figure 44: All agent 7 comparisons

12.2 Defective Agent 8 Performance

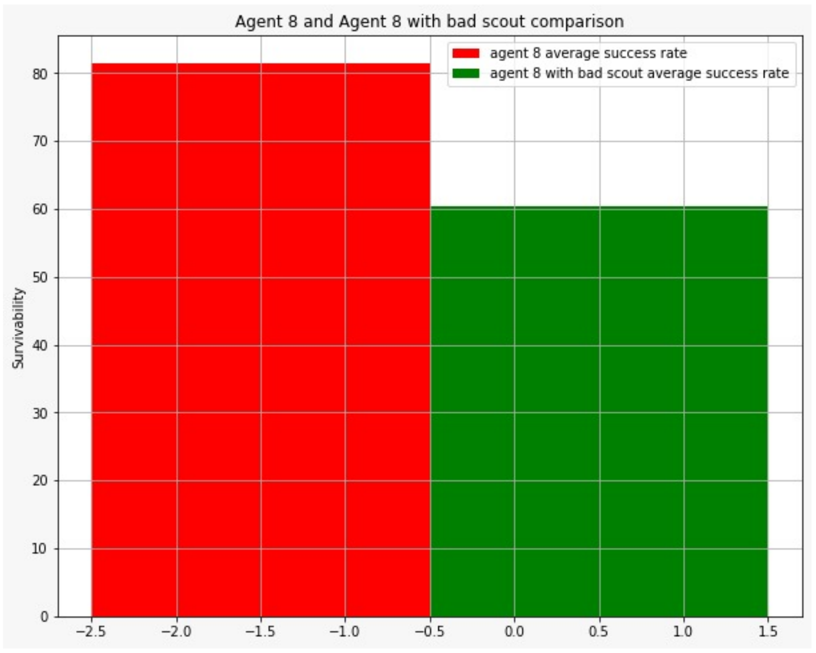


Figure 45: Agent 8 vs Agent 8 with bad scout Success Rate

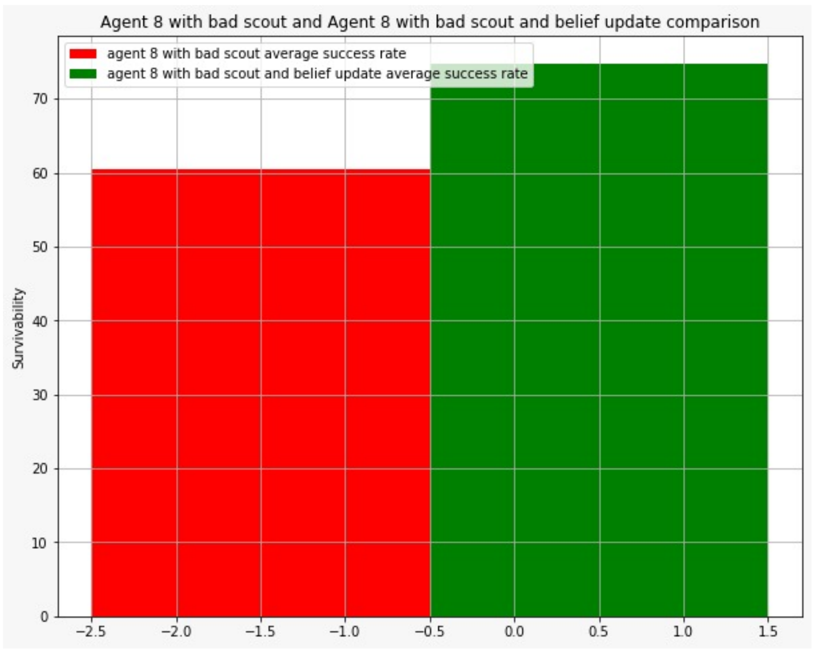


Figure 46: Agent 8 with bad scout vs Agent 8 with bad scout belief update

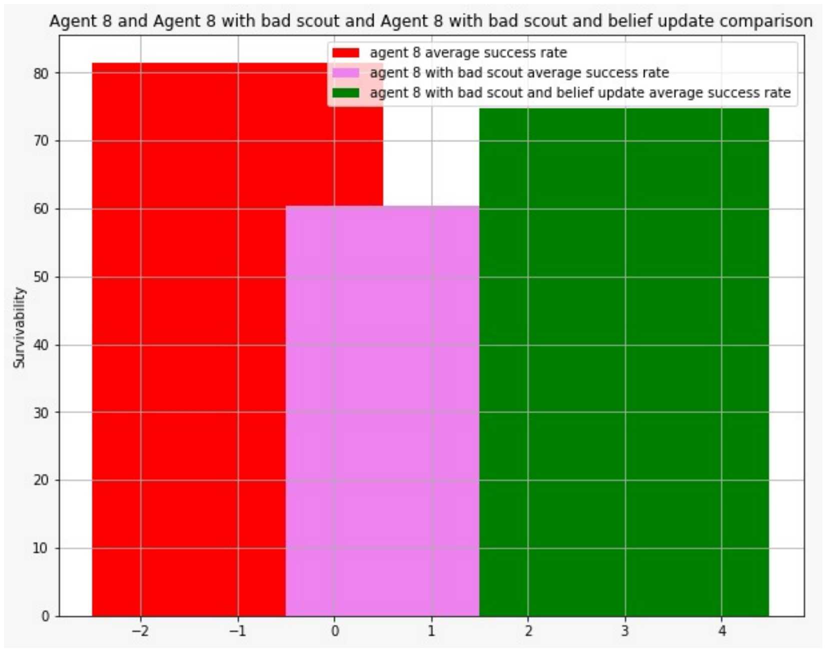


Figure 47: All agent 8 comparisons

13 Agent 9

In our partial information setting, we have built an agent that considers the next position in which the prey and predator can be before making a decision about the next move using our heuristic. (Our agent 8). As per our analysis on the failed cases, we have observed that our successful scout nodes is less and can be improved upon. We have also observed that since we dont know the location of both the prey and the predator, we couldn't make informed decisions. Therefore, since both the prey's and predator's movements are probabilistic, it is possible to build an agent that can outperform the above agent. That is, we can boot strap the agent 8 on our custom heuristic that will calculate the success rate and the average heuristic value for every agent's neighbours including the current node before making a decision. We can then analyse the pattern of the prey's and predators movements and based on the pattern returned by our heuristic, we can make more informed decisions.

14 Bonus

In the partial information setting, at each step, we scout or take a step but not both at the same time. To implement this kind of agent, there are a few takeaways that needs to be considered from all the above agents:

- On taking a move to a new node, it always gives us a new information about both the prey and predator. There are four possible cases for every step, the prey might be in the node, the predator might be in the node, both of them might be in the node or none of them might be in the node.
- For any of the first three cases, the game stops then and there. However, for the case 4, we get a new information about the prey and predator not being in the Agent's node.
- Using this new information about the prey and the predators not being in the moved cell, we can propagate this information into our belief arrays.

Therefore, when our predator's belief array is dense and our prey's belief array is dense/sparse, we can be certain about the fact that by taking the next step, we have a high chance of surviving. Hence, it is wise to take a step when our predator's belief array is dense. And in this case, if the prey's belief array is sparse, it helps us to make it denser or if the prey's belief array is dense, we end up making some progress towards the prey.

But, for the cases where the predator's belief array is sparse, that is we don't have a concrete information about the predator, it is wise to scout for a node and try to make our prey's and predator's belief arrays denser. There is also a special case when the prey's belief array becomes equal. In this case as well, it is wise to scout for a node to get new information about the prey and the predator.