The Circle of Life

Priyanshu Shrivastava (ps1276), Nikhil Tammina (nt444)

1 Introduction

This project places a significant emphasis on using a probabilistic model of the environment whenever there is uncertainty to assist in decision-making. Agent, Prey, and Predator are the three key components of this project. The main objective is to compel the agent to catch the prey before the predator captures the agent. Although there are predators that are moving towards the agent, the search algorithm we used, Dijkstra's, is a search algorithm that is used to find the shortest paths between nodes in the graph based on the predator's, agent's, and prey's positions information and makes the best possible decisions for the agent by dodging the predators and catching the prey. Every model today works like this agent, which deploys models in environments with different ranges of knowledge about the risks and asks them to find the best possible outcomes, eventually finding a robust solution.

2 Design Choices

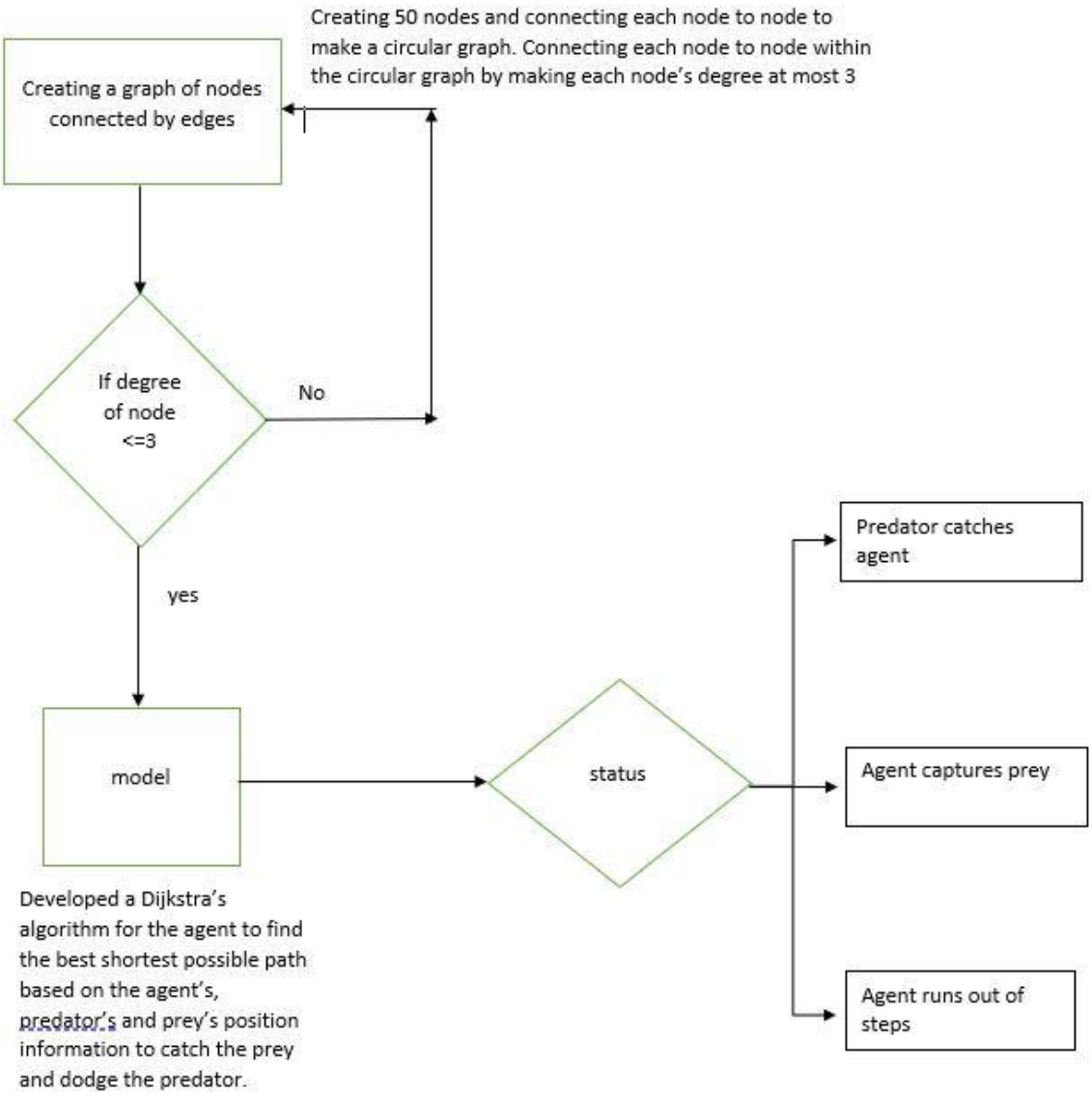


Figure - 1: Model Design

To begin with, we started the project by creating a graph of nodes numbered from 0 to 49 in a large circle. Now for each randomly picked node, we have added an edge to another node that is within 5 steps forward or backward along the primary loop. We are creating three major components, Agent, predator, and prey. Prey is spawned randomly in any node. We made sure that neither agent and predator nor agent and prey are spawned in the same node as the entire system loses or wins respectively if that's the case in the first iteration. We have designed Dijkstra's algorithm in order to find the shortest path between nodes. It finds the shortest path

from predator to agent and from agent to prey and the goal is to make sure that the agent avoids the predator based on different conditions of different agents and reaches the prey. Predators are easily distracted, it approaches the agents through nodes with a 60% probability and it moves randomly with a 40% probability. The probability of prey moving to its neighbors or staying in the same node is ¼. If the agent and predator are in the same node then the agent loses, and if the agent and the prey are in the same node then the agent wins.

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

If we add each edge from one node to the other node which isn't its neighbor and is within 5 nodes forward and backward, it will definitely be 25 edges which are the maximum edges. However, the least number of edges we could add is 18, as can be seen from the visual representation shown below. For example, let us consider the first fourteen nodes numbered from 1 to 14, leave the first, second, eighth, and ninth node with degree 2, and add the edges from the nodes numbered from 3 - 7 to 10 - 14. If we repeat the same operation till the 50th node, there will be 14 nodes left out. Hence we can reduce 7 edges from 25, which will lead to 18 edges which is the minimum we can add.

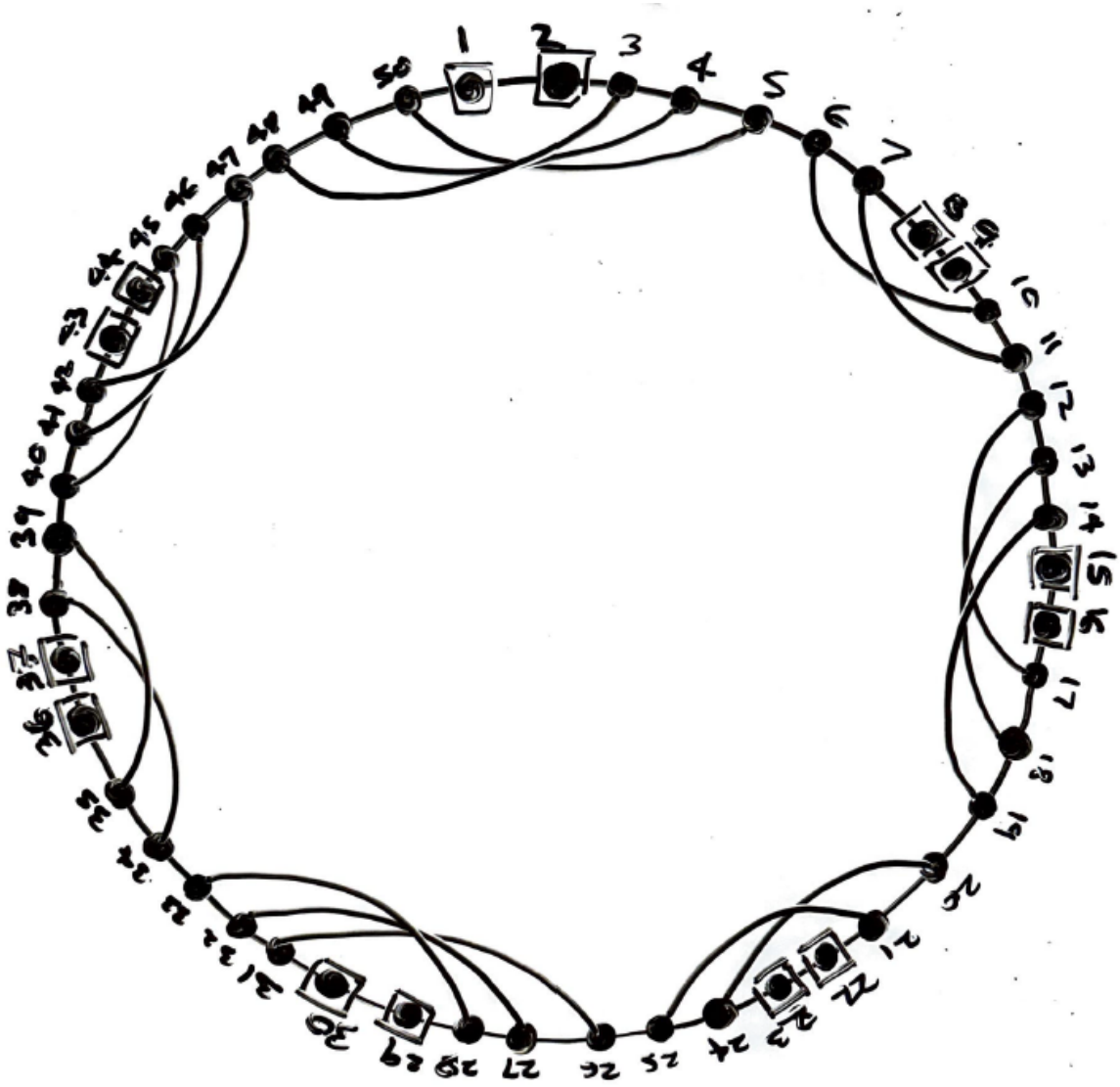


Figure - 2: Graph theory question

Agent 1

Here agent spawns at a random location at any node but with the condition that it doesn't spawn on the same node of predator or prey. We have designed Dijkstra's algorithm for agent 1 to find the shortest path from agent to prey. The order in which these components move in the network is agent, predator, and prey. A scenario of how Dijkstra looks -

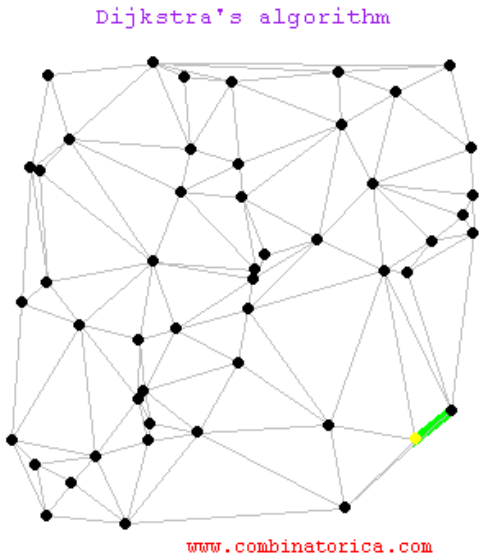


Figure - 3: Dijkstra

"Graph-notes/dijkstra.md at master · kjsce-codecell/Graph-notes." *GitHub*, https://github.com/kjsce-codecell/Graph-notes/blob/master/Algorithms/dijkstra.md. Accessed 14 November 2022.

```
# agents move
ndict[agent_pos]['state'] = -1
ndict[predator_pos]['state'] = -1
ndict[prey_pos]['state'] = -1
curr_dist_prey = dijkstra(agent_pos, prey_pos, clist)
curr_dist_predator = dijkstra(agent_pos, probable_predator_pos, clist)
neighbours = clist[agent_pos]
priority = list(np.zeros(len(neighbours)))
for x in range(len(neighbours)):
    n_dist_prey = dijkstra(neighbours[x], prey_pos, clist)
    n_dist_predator = dijkstra(neighbours[x], probable_predator_pos, clist)
    if len(curr_dist_prey) > len(n_dist_prey) and len(curr_dist_predator) < len(n_dist_predator):
        priority[x] = 1
    elif len(curr_dist_prey) > len(n_dist_prey) and len(curr_dist_predator) == len(n_dist_predator):
        priority[x] = 2
    elif len(curr_dist_prey) == len(n_dist_prey) and len(curr_dist_predator) < len(n_dist_predator):
        priority[x] = 3
    elif len(curr_dist_prey) == len(n_dist_prey) and len(curr_dist_predator) == len(n_dist_predator):
        priority[x] = 4
    elif len(curr_dist_predator) < len(n_dist_predator):
        priority[x] = 5
    elif len(curr_dist_predator) == len(n_dist_predator):
        priority[x] = 6
    else:
        priority[x] = 7
priority_max = min(priority)
move_to_list = []
for y in range(len(priority)):
    if priority[y] == priority_max:
        move_to_list.append(neighbours[y])
if priority_max == 7:
    agent_pos = agent_pos
else:
    agent_pos = random.choice(move_to_list)
```

Figure - 4: Agent 1 code snippet

The agent initially surveys its neighbors and gives priority to the neighbors in the following order:

-Neighbors that are closer to the Prey and farther from the Predator.

As we can see from the above code snippet the agent's neighboring nodes' distance from prey is less than the agent's current node from prey and the agent's current distance from predator is less than the distance between the agent's neighboring nodes and predator.

-Neighbors that are closer to the Prey and not closer to the Predator.

From the above code snippet, the agent’s neighboring nodes' distance from prey is less than the agent’s current node from prey, and the agent’s current distance from the predator is equal to the distance between the agent’s neighboring nodes and predator.

-Neighbors that are not farther from the Prey and farther from the Predator.

From the above code snippet, the agent’s neighboring nodes' distance from prey is equal to the agent’s current node from prey, and the agent’s current distance from predator is less than the distance between the agent’s neighboring nodes and the predator.

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

From the above code snippet, the agent’s neighboring nodes' distance from prey is equal to the agent’s current node from the prey, and the agent’s current distance from the predator is equal to the distance between the agent’s neighboring nodes and the predator.

-Neighbors that are farther from the Predator.

This case from the above code snippet only considers the agent’s neighboring nodes' distance from the predator which is greater than the agent’s current node from the predator.

-Neighbors that are not closer to the Predator.

This case from the above code snippet only considers the agent’s neighboring nodes' distance from the predator which is equal to the agent’s current node from the predator.

-Sit still and pray.

This is the last order in this priority where the agent does not move.

Figure - 5: Agent 1 Analysis

Success rate:87.4%

We can infer the success rate by looking at the above graph, the agent managed to avoid the predator and capture the prey in most of the iterations. However, in graph number 7, the predator captured the agent for approximately 35 times out of 100 iterations which is the least success rate compared to other graph nodes. We can find the other least success rates at 23rd graph number. Hence in order to increase the success rate, we have designed agent 2 with our own rules which we have explained briefly in the next page.

Agent 2

To make agent 2 better than agent 1, we initially considered the same priority order as agent 1, and we have taken the distance from the agent's neighboring nodes to the prey's neighboring nodes instead of the prey's current node. The agent will choose its neighboring node based on the average distance from the prey's neighboring nodes. However, the success rate is just 1-3% better than agent 1's if we follow this method. We found a better way to increase the success rate by making changes to the agent's movements (priority order).

```
# agents move
ndict[agent_pos]['state'] = -1
ndict[predator_pos]['state'] = -1
ndict[prey_pos]['state'] = -1
curr_dist_prey = dijkstra(agent_pos, prey_pos, clist)
curr_dist_predator = dijkstra(agent_pos, predator_pos, clist)
neighbours = clist[agent_pos]
priority = list(np.zeros(len(neighbours)))
for x in range(len(neighbours)):
    n_dist_prey = dijkstra(neighbours[x], prey_pos, clist)
    n_dist_predator = dijkstra(neighbours[x], predator_pos, clist)
    if predator_pos in n_dist_prey:
        priority[x] = 6
    elif predator_pos in n_dist_prey and len(n_dist_prey) == 2 * len(n_dist_predator):
        priority[x] = 7
    if len(curr_dist_prey) > len(n_dist_prey) and len(curr_dist_predator) < len(n_dist_predator):
        priority[x] = 1
    elif len(curr_dist_prey) == len(n_dist_prey) and len(curr_dist_predator) < len(n_dist_predator):
        priority[x] = 2
    elif len(curr_dist_prey) < len(n_dist_prey) and len(curr_dist_predator) < len(n_dist_predator):
        priority[x] = 3
    elif len(curr_dist_predator) == len(n_dist_predator):
        priority[x] = 4
    else:
        priority[x] = 5
print(priority)
priority_max = min(priority)
move_to_list = []
for y in range(len(priority)):
    if priority[y] == priority_max:
        move_to_list.append(neighbours[y])
if priority_max == 5:
    agent_pos = agent_pos
else:
    agent_pos = random.choice(move_to_list)
```

Figure - 6: Agent 2 code snippet

As we can see from the snippet above, the priorities in which the distances are considered are different. Firstly we are assigning the worst priority (priority 1 being the highest and 7 the lowest) to the nodes in which the predator is in the path of the prey (priority 6). Secondly, we want the distance from neighboring nodes to the predator to always be more than the distance of the current node to the predator so that we are always moving away from the predator increasing the survivability. We have removed the cases where the distance from the neighboring node to the predator is less than the distance from the current node to the predator as it only increases the probability of the predator getting close to the agent and hinders survivability. Our success rate increased by 10-12%.

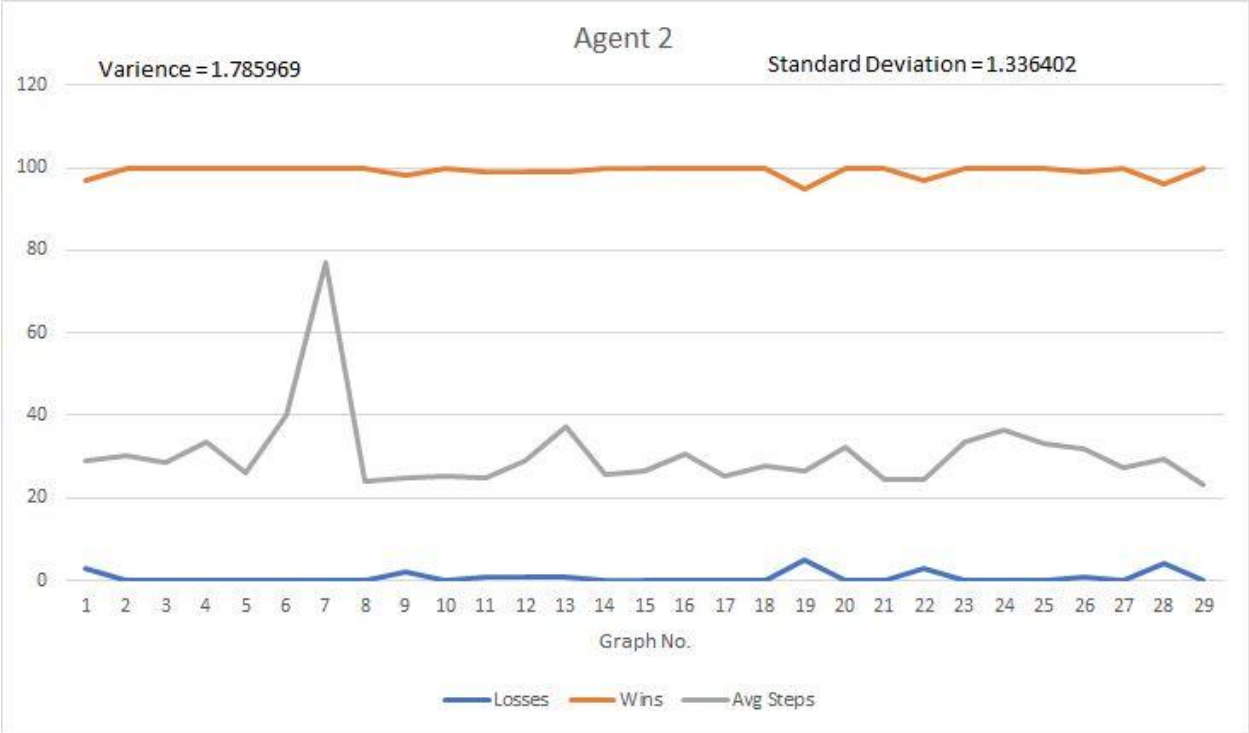


Figure - 7: Agent 2 Analysis

Success rate: 99.33%

From the graph, we can clearly and safely state that agent 2's success rate is the highest. The agent is successfully dodging the predator and capturing the prey most of the time. However, in graph number 7, even though it has taken many steps compared to the other graph numbers, it successfully managed to capture the prey avoiding predator in all 100 iterations. Predator captured the agent, making the agent unsuccessful in its task 5 times out of 100 in graph number 19, which is the least success rate compared to all other iterations.

Agent 3

Agent 3 contains information about only where the predator is but not the prey. Whenever the agent moves, it surveys the entire network based on the belief system and determines whether the prey exists in that node or not. It technically uses a drone system to figure out whether a prey exists in that node, and it updates the belief system accordingly. The belief system includes the probability that the prey exists in that node. Whenever the agent moves in a node and if the prey doesn't exist, it divides the probability uniformly among all the other nodes that the agent hasn't visited yet making sure the sum of all beliefs is 1. Whenever the drone finds the prey in a particular node then it updates the probability of that node as 1 and the remaining nodes as 0.

Q In these cases, how should you be updating the probabilities of where the Prey is?

As I have already mentioned above, the belief contains the probability of each node in the network, to check whether there is prey in a particular node we send a drone initially to check the existence of prey. We defined the method prey_belief_target which assigns the priority to which nodes to be checked by the drone first. So, firstly, it iterates through the belief array consisting of the probability of the entire nodes in the network and chooses the node with the maximum probability of a node containing prey. If there are multiple nodes that contains the same maximum probability we will break the ties at random. If the drone moves to a particular node and finds the prey, then it updates the probability of that node as 1 and the remaining nodes as 0. If the next node in which the drone moves doesn't contain the prey then it makes that particular node probability 0 and uniformly distributes the previous probability of that node to the other nodes and updates the belief system. The formulae we have used for the agent movement and drone movement check is

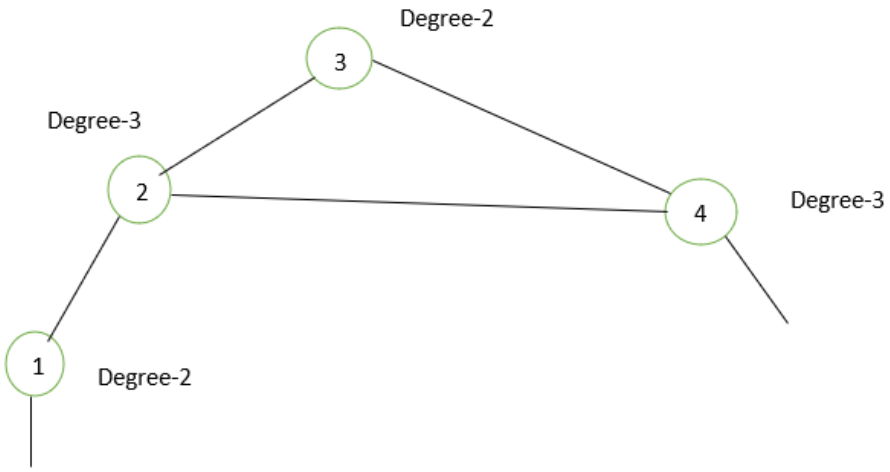
$$\text{Probability of node in the network} = \frac{\text{Probability of the current node}}{1 - \text{previous probability of the node surveyed}}$$

After beliefs are updated by drones, the agent starts to move through nodes searching for prey, it repeats the same operations as drone but it surveys only its neighbors and updates the beliefs

accordingly by the same formulae mentioned above. It keeps updating the beliefs until it finds the prey and once it found the prey, then it makes that node's probability 1 and the remaining 0.

The probability of the agent considering the prey's movement and updating the belief system is explained explicitly below:

Instead of considering all the 50 nodes, let us consider the following scenario for simplicity:



Let us consider prey is located on node 2, let us consider each node above contains a probability of 1/4. Now the belief of node 2 for the existence of prey will be updated in the following way:

$$\text{Belief}(2) = \frac{P(2)}{D(2)+1} + \frac{P(3)}{D(3)+1} + \frac{P(4)}{D(4)+1} + \frac{P(1)}{D(1)+1}$$

(Where D represents degree)

$$= \frac{1/4}{3+1} + \frac{1/4}{2+1} + \frac{1/4}{2+1} + \frac{1/4}{2+1}$$

$$= 5/16$$

So the probability of prey at being node 2 is 5/16 and the belief gets updated.

Now For all nodes (i=1 to n) in the network, the same Belief(i) function will be updated.

Now we could generalize a Belief update formulae for the prey for i = 1 to n nodes in the network as

$$P(i) = \frac{P(i)}{D(i)+1} + \frac{P(\text{ith neighbor 1})}{D(\text{ith neighbor 1})+1} + \frac{P(\text{ith N 2})}{D(\text{ith N 2})+1} + \frac{P(\text{ith N 3})}{D(\text{ith N 3})+1}$$

For different i for different nodes, the entire probability for all the nodes gets updated to a belief system based on the belief formulae.

We implemented the calculation for the belief update which was explained above in our code as follows:

```
def prey_belief_update(node, clist, belief, prey_pos, case):
    new_belief = belief.copy()
    if case == "agent_move":
        # distribution = belief[node] / (len(belief) - (count + 1))
        for x in range(len(belief)):
            new_belief[x] = belief[x] / (1 - belief[node])
        new_belief[node] = 0

    if case == "drone_check":
        if node == prey_pos:
            for x in range(len(belief)):
                new_belief[x] = 0
            new_belief[node] = 1
        else:
            # distribution = belief[node] / (len(belief) - (count + 1))
            for x in range(len(belief)):
                new_belief[x] = belief[x] / (1 - belief[node])
            new_belief[node] = 0

    if case == "prey_move":
        for x in range(len(belief)):
            neighbours = clist[x]
            neighbours_belief = 0
            for neighbour in neighbours:
                neighbours_belief += belief[neighbour] / ((len(clist[neighbour])) + 1)
            new_belief[x] =  (belief[x] / (len(clist[x]) + 1)) + neighbours_belief
    return new_belief
```

Figure - 8: prey belief update

Figure - 9: Agent 3 Analysis

Success rate: 89.53%

In the above graph, even though we don't have the prey's information or whereabouts, we get the success rate as 89.53% which is convincing. The success rates for all the graph numbers vary between 80% and 100% which is consistent.

Agent 4

For creating this agent, we have used agent 2's logic and agent 3's belief system. In a partial prey information setting, where the agent will not have any trace of the prey's position. Based on agent 4's logic, it will avoid the predator and keeps on updating the belief system as per agent 3. It works on the concept dying is worse than not finding prey.

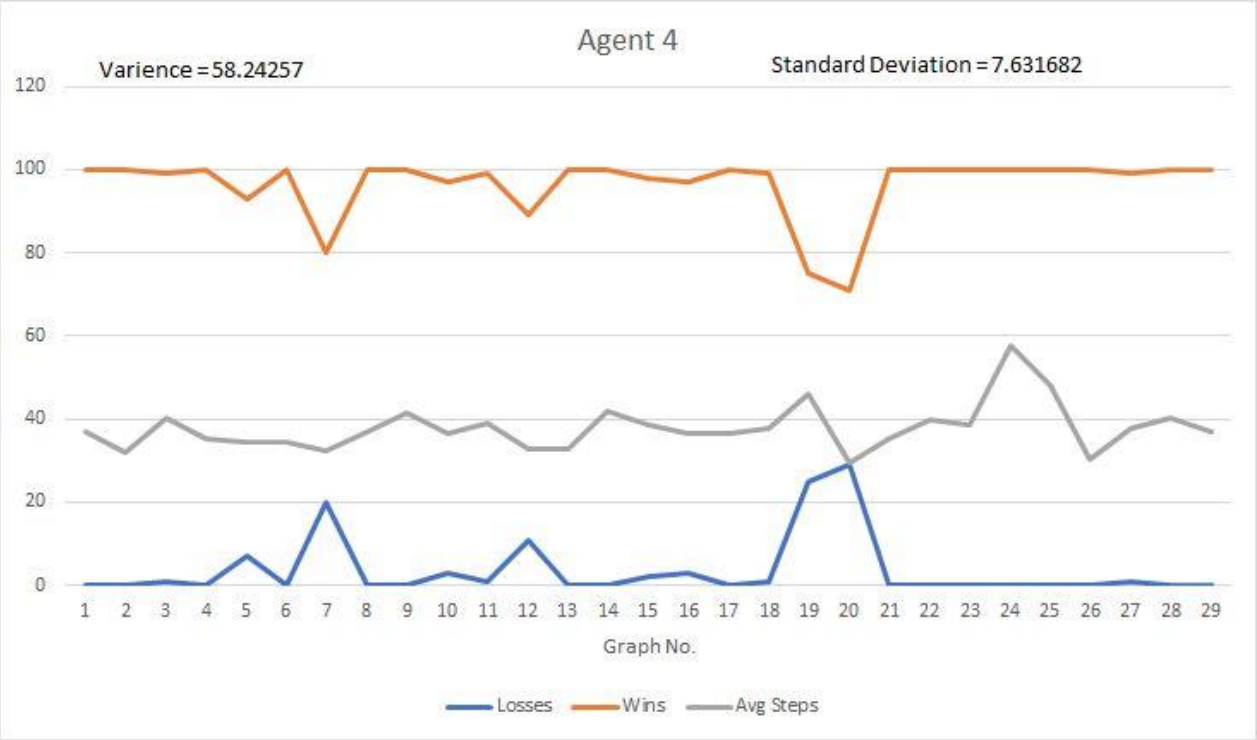


Figure - 10: Agent 4 Analysis

Success rate: 96.03%

Agent 4's success rate is satisfying, even though it doesn't contain prey's information its results are astonishing, The graph shows that from graph numbers 18 to 21, the success rate decreased from the normal. However, apart from that, it stayed in the 90s success rate for all the iterations.

Agent 5

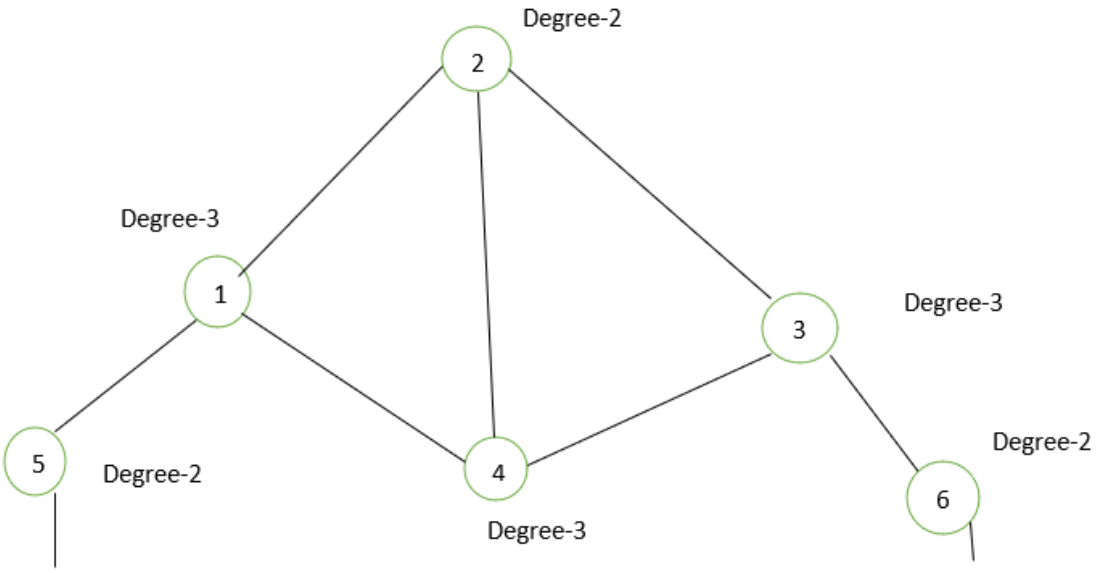
For this agent, it doesn't contain the whereabouts of the predator, it only consists of the prey's information. Whenever the agent surveys any node, it updates the information of the nodes that it already visited and if it doesn't contain the predator in it, it will update the belief system. After the belief system is updated we need to figure out the node that needs to be surveyed, for that we iterate through the belief array and find the maximum probability among all the nodes, and if there are multiple nodes with the same maximum probability we break the tie by calculating the distances from the each of the node to the agent and select the one with minimum distance and again if there are multiple nodes with the same distance we break the tie at random. The probability that the belief system is getting updated for the drone and the agent is

$$\text{Probability of node in the network} = \frac{\text{Probability of the current node}}{1 - \text{previous probability of the node surveyed}}$$

How do the probability updates for the Predator differ from the probability updates for the Prey?

The probability of the agent considering the predator's movement and updating the belief system is explained explicitly below:

Let's consider the below case for simplicity



$$\text{Distracted\_probability}(2) = \frac{P(1)}{D(1)} + \frac{P(4)}{D(4)} + \frac{P(3)}{D(3)}$$

$$\text{Distracted\_probability\_final}(2) = \text{Distracted\_probability}(2) * 0.4$$

$$\text{Focused\_probability}(2) = \frac{P(1)}{1} + \frac{P(4)}{2}$$

$$\text{Focused\_probability\_final}(2) = \text{Focused\_probability}(2) * 0.6$$

$$\text{Total\_probability}(2) = \text{Distracted\_probability\_final}(2) + \text{Focused\_Probability\_final}(2)$$

Lets understand the above calculation:

Here, we are iterating through all the node's belief values and updating when the predator is moving. For node 2 to be updated to its new belief probability, we know that the predator is distracted, so the distracted probability is first calculated by considering all the neighbors of that node and adding the probability of each neighbour divided by the degree of that neighbor as the predator is distracted and can travel to any of the neighboring node 40% of the time.

For the rest 60% the predator is focused and only considers the neighbors whose distance from the agent is smallest. In this case, it will check the distance from the neighbour's neighbor to the agent and finds the list of nodes with the shortest distance, and checks if the node being updated is part of that list. Let's consider the above example for node 2. It has 3 neighbours, node 1, node 4, and node 3, so for node 4 to distribute its probability to node 1 it will check the distance from node 1, node 2 and node 3 to agent's position. It finds that node 2 and node 3 gives the shortest path and node 2 is part of the shortest distance list, so it distributes its probability equally among node 2 and node 3, hence P(4)/2 and for node 1 it only has neighbour node 1 which has the shortest distance to agent, hence P(1)/1. But node 3 has the shortest distance from node 6 and node 2 is not part of that shortest distance list, hence node 3 entirely distributes its probability to node 6 and nothing gets added to node 2.

This is how we incorporated the above logic in our code for belief update of the predator:

```
# updating the belief system of predator
def predator_belief_update(node, clist, belief, predator_pos, case):
    new_belief = [0] * 50
    if case == "agent_move":
        if belief[node] == 1:
                return belief
        for x in range(len(belief)):
            new_belief[x] = belief[x] / (1 - belief[node])
        new_belief[node] = 0

    if case == "drone_check":
        if belief[node] == 1:
                return belief
        if node == predator_pos:
            for x in range(len(belief)):
                new_belief[x] = 0
            new_belief[node] = 1
        else:
            for x in range(len(belief)):
                new_belief[x] = belief[x] / (1 - belief[node])
            new_belief[node] = 0

    if case == "predator_move":
        for x in range(len(belief)):
            neighbours_x = clist[x]
            distracted_prob = 0
            undistracted_prob = 0
            for neighbour in neighbours_x:
                distracted_prob += (belief[neighbour]/(len(clist[neighbour])))
                neighbours_y = clist[neighbour]
                dist_n = []
                for n in neighbours_y:
                    dist = dijkstra(n, node, clist)
                    dist_n.append(len(dist))
                min_dist = min(dist_n)
                small_dist_nodes = []
                for i in range(len(dist_n)):
                    if dist_n[i] == min_dist:
                        small_dist_nodes.append(neighbours_y[i])

                if x in small_dist_nodes:
                    undistracted_prob += (belief[neighbour])/len(small_dist_nodes)
            new_belief[x] = (distracted_prob) * 0.4 + (undistracted_prob) * 0.6
    return new_belief
```

Figure - 11: Predator belief update

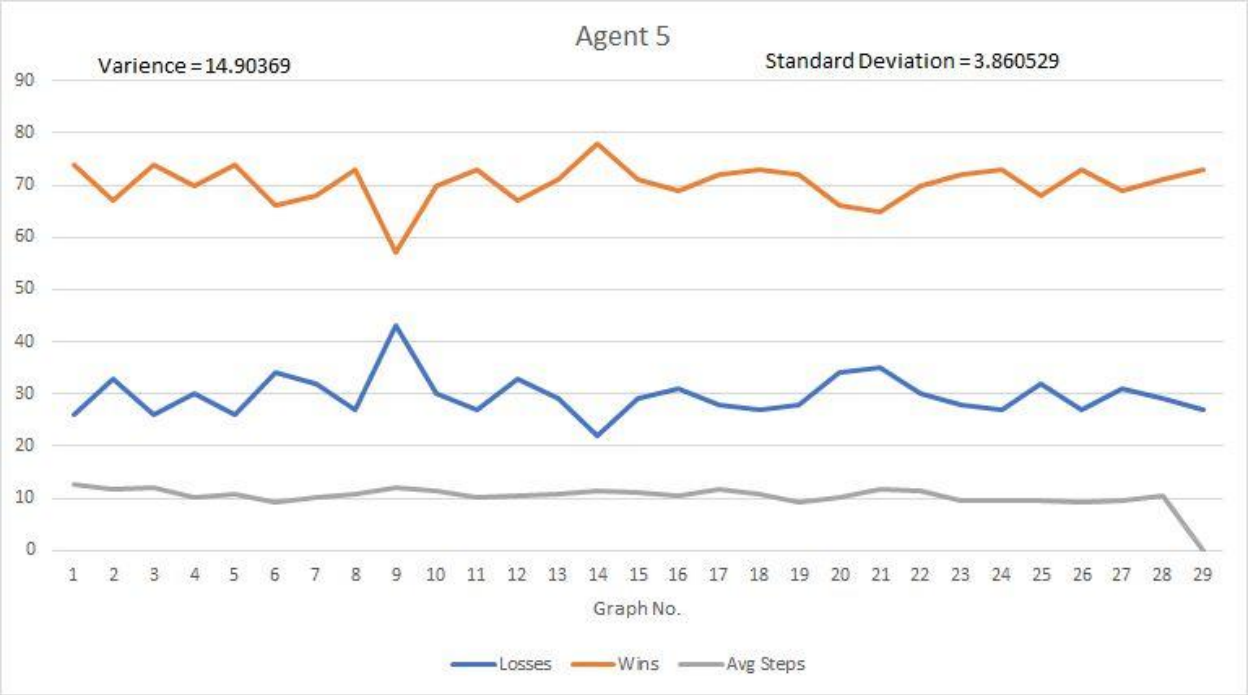


Figure - 12: Agent 5 Analysis

Success rate: 70.4%

Since agent 5 will not contain predator's information, it will be difficult for it to avoid the predator as effectively as it avoided in case of previous agents. Hence the success rate is somewhat low comparatively. It is explicit that the success rate varies only between 60 and 80. However, in graph number 9, out of 100 iterations only 57 are successful.

Agent 6

To make agent 6 we have used agent 2's movement logic and agent 5's predator's belief update as we don't have the exact information of where the predator is.

Figure - 13: Agent 6 Analysis

Success rate: 85.5%

In the above graph, the chances of agent getting caught by predator is comparatively low. We are getting the result of agent losing or winning in approximately 11 steps for almost all graph numbers. The success rates vary from 80 to 90 except for the graph numbers 15 and 21. For partial predator settings, this is the maximum success rate we could achieve.

Agent 7

In this agent, we don't know either the predator's or prey's information in the network. So, in this case, we have used agent 3's and agent 5's belief logic. We update the prey's belief as well as the predator's belief. An agent can first select any node in the graph to survey. In this scenario, the Agent must monitor the belief states of both the Predator and the Prey, We are updating the belief of the predator when drone surveys, when the agent moves, and when predator moves in accordance with the belief update of the agent 5, similarly we update belief of prey when drones surveys, when agent moves, and when prey moves. The probable positions of prey and predator are extracted using the method prey_belief_target and method predator_belief_target respectively.

Figure - 14: Agent 7 Analysis

Success rate:60%

As we don't have the predator's nor prey's whereabouts, the agent's success rate seems to be 60%. From the graph, the agent's success rate of capturing the prey and avoiding the predator varies only between 50 and 70%. However, in the 8th graph number, the success rate of the agent exceeds the failure rate just by approximately 5%. The probability of the agent dodging the predator is still greater than getting caught by it.

Agent 8

This agent is an improvised version of agent 7 where it uses the belief updates of prey and the predator of agent 7 and uses agent 2's logic for the improvised movement of the agent. As sometimes agent 7 gives priority to the movement where it not moving away from the predator, the survivability of the agent decreases. In agent 8, it is using agent 2's logic of moving away from the predator majority of the time. The survivability of the agent is in less danger compared to 7.

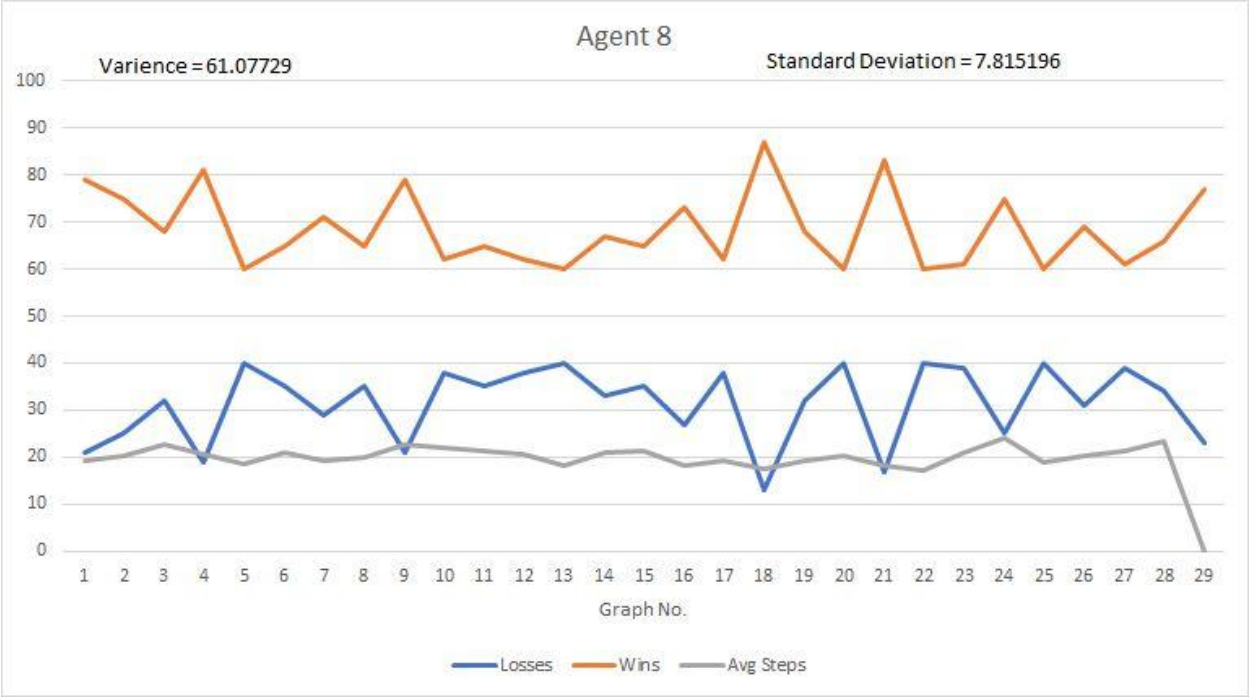


Figure - 15: Agent 8 Analysis

Success rate: 68.33%

We can see that all even-numbered agents have improvised the result compared to the odd number of agents significantly.

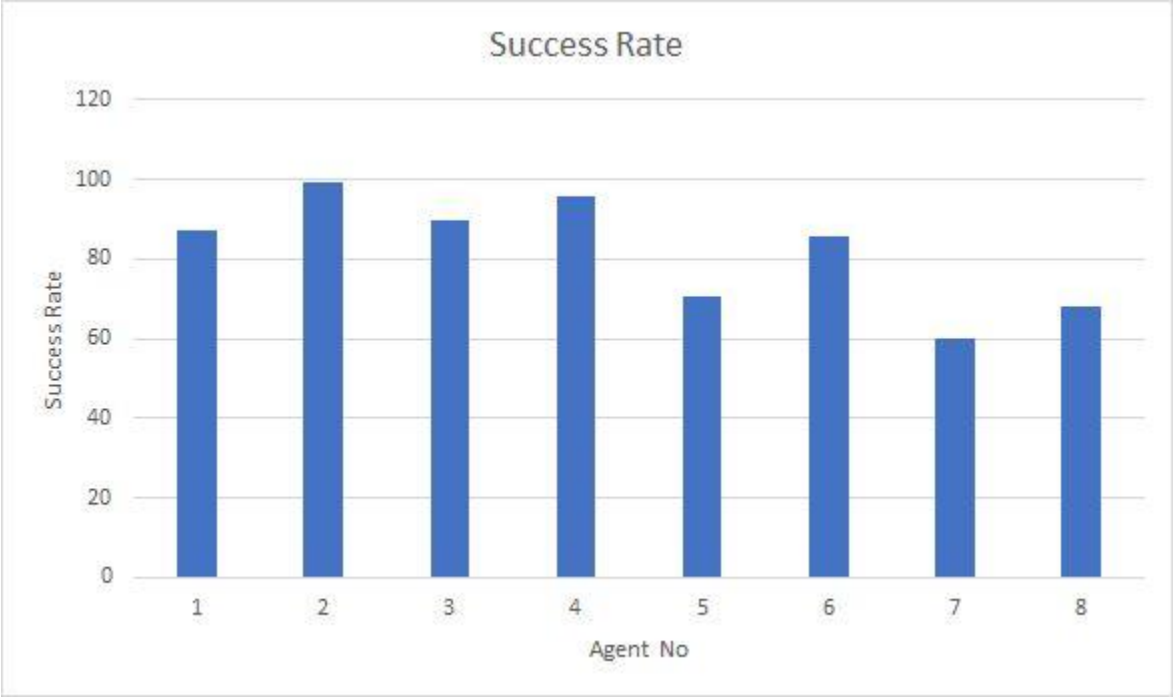


Figure - 16: Success rates of all the agents

We can infer from the graph that all the even-numbered agents are performing better than it's predecessor's odd agents.

Faulty drone setting

In this environment setting, the drone has a defect where it sometimes (10% of the times) gives feedback of the node being empty when it is actually being occupied. This basically means the drone is giving us false negative output which results in agents being confused whether the node being actually empty or being occupied by prey or predator. This is implemented by making changes in belief update, where instead of making the probability of the node containing prey or predator as 1 or 0, it only makes that node equal to one 90% of the time and for the rest 10% it makes the probability of that node 0 and redistributes the previous probability to all the other nodes.

Agent 7A

Agent 7A is the same as agent 7 but it faces the problem of a faulty drone in which the drone doesn't give proper information about the existence of predator or prey. It will show that the node in a graph is empty even though a predator or prey exists in that particular node. So there is a 0.1 probability (10% chance) that the drone gives a false negative update. So, because of this scenario, the agent gets confused which leads to possible distractions ultimately leading to a low success rate as shown below. In the following way, we have incorporated the faulty drone setting in the system.

```
if case == "drone_check":
    if belief[node] == 1:
            return belief
    if node == predator_pos:
        list_prob = list(np.arange(1,10))
        prob = random.choice(list_prob)
        if prob <10:
            new_belief = [0] * 50
            new_belief[node] = 1
        else:
            for x in range(len(belief)):
                new_belief[x] = belief[x] / (1 - belief[node])
            new_belief[node] = 0
    else:
        for x in range(len(belief)):
            new_belief[x] = belief[x] / (1 - belief[node])
        new_belief[node] = 0
```

Figure - 16: faulty drone setting code snippet

Figure - 17: Agent 7A Analysis

Success rate: 48%

From the graph above, it is explicit that the loss rate surpasses the success rate in most of the iterations. To overcome this problem we have developed Agent 7B.

Agent 8A

Agent 8A is the same as agent 8 but it faces the problem of a faulty drone in which the drone doesn't give proper information about the existence of predator or prey. Even though a predator or prey is present in that specific node of a network, it might demonstrate that the node is empty. Therefore, there is a 10% possibility (0.1 probability) that the drone may provide a false negative update. As a result of this circumstance, the agent becomes confused, which may generate diversions and, eventually, a low success rate, as illustrated below.

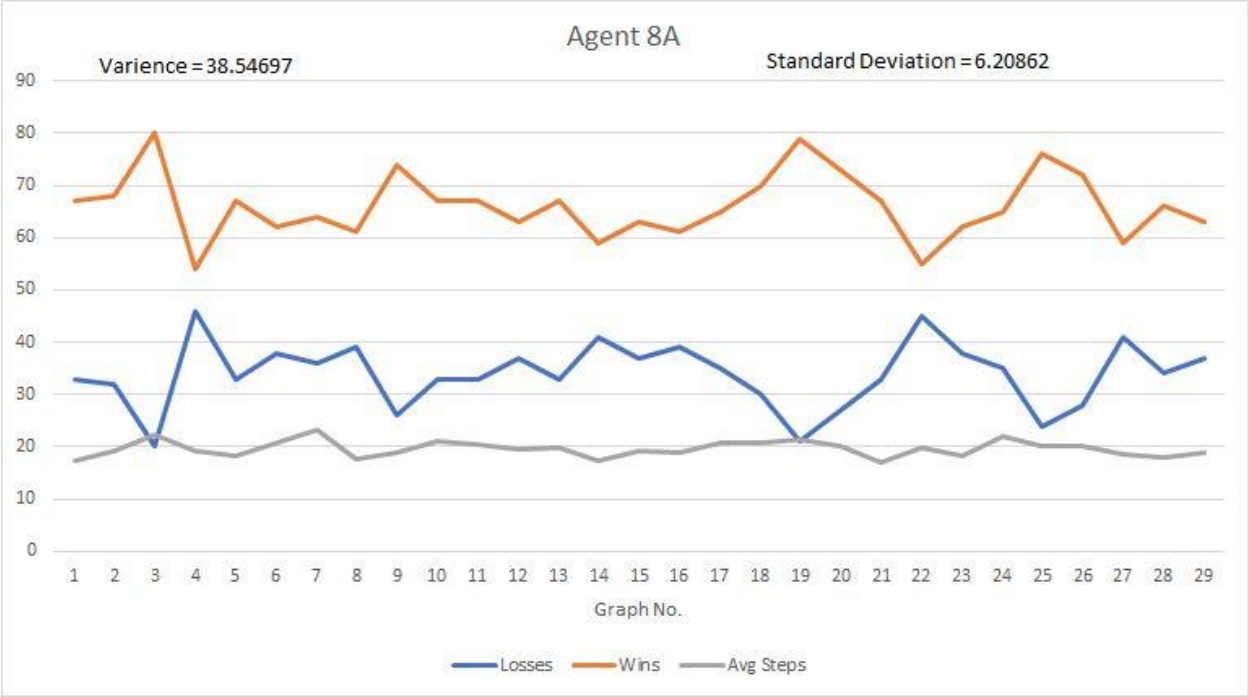


Figure - 18: Agent 8A Analysis

Success rate: 63.4%

In the graph above, most of the success rate varies between 60 and 80 except for very few graph numbers. To overcome this problem, we have designed agent 8B.

Agent 7B

Agent 7B possibly overcomes the faulty drone setting. Whenever the drone goes to a node and surveys it, when it finds that there is no prey or predator in it, instead of updating the belief as 0, it updates the 10% chance that there is prey or predator in that node. Then, it equally divides the node's previous probability to the rest of the nodes with 0.9 * node's previous probability.

This is how we have improvised faulty drone setting using the logic previously described:

```
if case == "drone_check":
    if belief[node] == 1:
            return belief
    if node == prey_pos:
        list_prob = list(np.arange(1,10))
        prob = random.choice(list_prob)
        if prob <10:
            new_belief = [0] * 50
            new_belief[node] = 1
        else:
            for x in range(len(belief)):
                new_belief[x] = belief[x] / (1 - (0.9 * belief[node]))
            new_belief[node] = 0.1
    else:
        for x in range(len(belief)):
            new_belief[x] = belief[x] / (1 - belief[node])
        new_belief[node] = 0
```

Figure - 19: solution for faulty drone setting code snippet

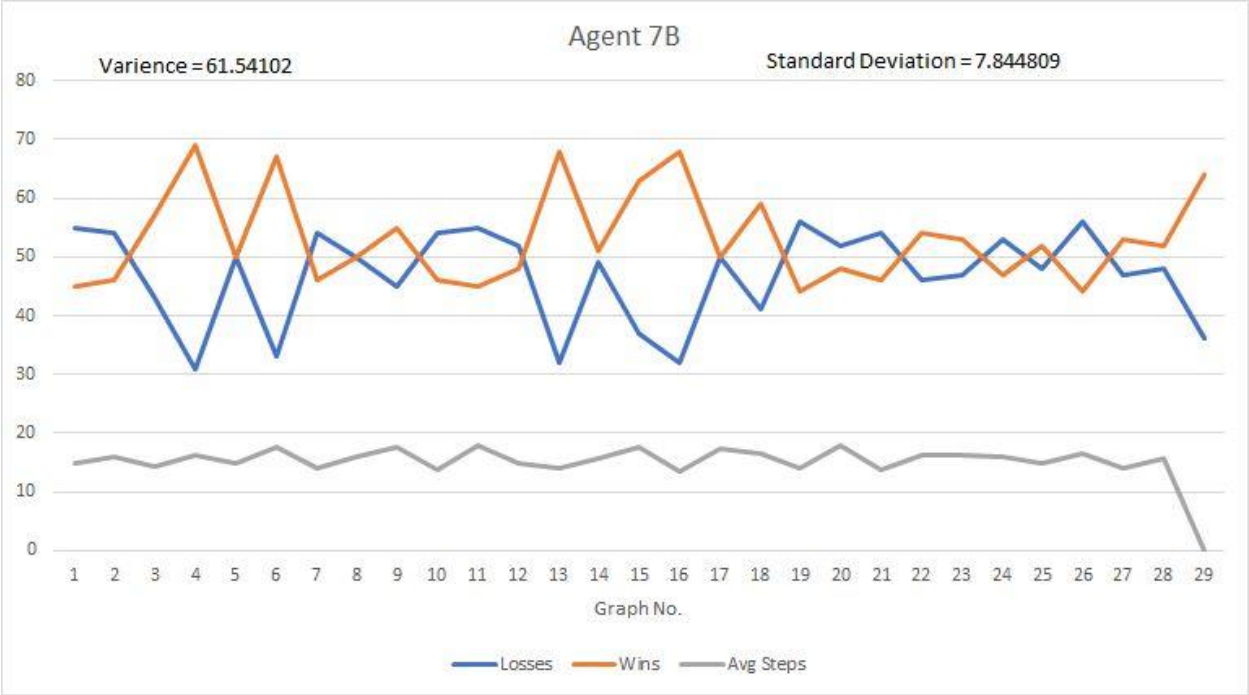


Figure - 20: Agent 7B Analysis

Success rate: 53.1%

From the above graph, we can say that the agent 7B's success rate increased from agent 7A's. The maximum success rate that this agent is going for is 70% while the minimum is 45%.

Agent 8B

Agent 8B possibly overcomes the faulty drone setting. Whenever the drone goes to a node and surveys it, when it finds that there is no prey or predator in it, instead of updating the belief as 0, it updates the 10% chance that there is prey or predator in that node. Then, it equally divides the node's previous probability to the rest of the nodes with 0.9 * node's previous probability. Agent 8B has a better success rate than agent 7B because it also considers agent 2's logic for the agent's movement.

Figure - 21: Agent 8B Analysis

Success rate: 66.2%

Agent 9

Can we make an agent that can do better than Agent 7B and Agent 8B?

We have a belief that is getting hindered by the faulty drone which is the reason for agent being deceived. This is when the drone comes back with the information that the node is empty. To overcome this we can think of the belief system to be partially informative and only use it when the we have the surety that information being retrieved is actually true. When the drone is providing the information of any node and if it finds prey or predator we know that drone is not lying and can successfully use this information and on the contrary when it finds the node is empty the agent cannot be sure of this information. In the scenario, when the drone finds that the node is empty, if we consider the previous belief and make the decision according to the previous belief assumption until the drone actually finds prey or predator, we might increase the survivability and success rate.