Artificial Intelligence Project 2 (CS520)

The Circle of Life

Viswajith Menon (VM623)

Zeeshan Ahsan (ZA224)

The Environment –

The environment consists of a circular graph of 50 nodes numbered (1-50).
Initially every node in the graph has a degree of 2 and is connected to its neighbor with the following logic –
Node 1 – Connected to node 50 and node 2.
Node 50 – Connected to node 1 and node 49.
Node i (i between 2,49) – Connected to node (i-1) and node (i+1) (2<=i<=49).

Once the initial graph is set up, we pick a random node and connect it to a node within 5 steps ahead of it or behind it. This step is repeated until all nodes have a degree of 3 or no other edges can be added with the conditions provided.

Implementation of the environment –

Graph –

The graph's vertices and edges are stored in the form of an adjacency list. We have used a dictionary to maintain the adjacency list, where the 'key' is the vertex, and the 'value' is a list of vertices it is connected to.

Initially connecting the adjacent vertices to form the circular graph –

```
temp=1
adjacency_list={}
for i in range(1,no_of_nodes+1):
    if(i==1):
        adjacency_list[temp]=[no_of_nodes,temp+1]
    elif(i==no_of_nodes):
        adjacency_list[temp]=[temp-1,1]
    else:
        adjacency_list[temp]=[temp-1,temp+1]
    temp=temp+1
```

Now to create the extra edge with the conditions given, we have created a dictionary called 'degree_check' which will be a copy of the originally created 'adjacency_list'.

We then pick a node at random and check the available nodes it can create an edge with.

If no node is available, the degree of the node will remain 2 and it is removed from ‘degree_check.’
If a possible node is found, the ‘adjacency_list’ updates both these nodes and then both are removed from ‘degree_check’.
This process is repeated until no nodes are left in the ‘degree_check’ dictionary.

```
while(degree_check):
    choice = np.random.choice(index)
    l= list(range(choice-5,choice+5+1))
    list_of_possible_edges=[]
    for x in l:
        if(x==0):
            continue
        if(x<1):
            if(no_of_nodes+1+x in index):
                list_of_possible_edges.append(no_of_nodes+1+x)
            else:
                continue
        elif(x>no_of_nodes):
            if(x-no_of_nodes in index):
                list_of_possible_edges.append(x-no_of_nodes)
            else:
                continue
        else:
            list_of_possible_edges.append(x)
    list_of_possible_edges = list(dict.fromkeys(list_of_possible_edges))
    list_of_possible_edges = [ x for x in list_of_possible_edges if len(adjacency_list[x])==2]
    list_of_possible_edges.remove(choice)
    if(choice-1>0 and choice-1 in list_of_possible_edges):
        if(choice==1):
            list_of_possible_edges.remove(no_of_nodes)
        else:
            list_of_possible_edges.remove(choice-1)
    if(choice+1<=no_of_nodes and choice+1 in list_of_possible_edges):
        if(choice==no_of_nodes):
            list_of_possible_edges.remove(1)
        else:
            list_of_possible_edges.remove(choice+1)
    #print("list_of_possible_edges ",list_of_possible_edges)
```

```
if(len(list_of_possible_edges)==0):
    index.remove(choice)
    del degree_check[choice]
else:
    chosen_edge=np.random.choice(list_of_possible_edges)
    #print("chosen_edge ",chosen_edge)
    if(len(adjacency_list[chosen_edge])==2):
        adjacency_list[chosen_edge].append(choice)
        adjacency_list[choice].append(chosen_edge)
        count=count+1
        #print("adjacency_list[choice] ",adjacency_list[choice])
        #print("adjacency_list[chosen_edge] ",adjacency_list[chosen_edge])
        flag=1

if(flag==1):
    index.remove(choice)
    index.remove(chosen_edge)
    flag=0
    del degree_check[choice]
    del degree_check[chosen_edge]
```

Predator and Prey –

The prey's movements are random. At every timestep it chooses either to stay in its current position or move to one of its neighbors with equal probability.
When the predator moves, it looks at its available neighbors, and calculates the shortest distance to the agent for each neighbor it can move to. It then moves to the neighbor with shortest distance to the agent.

Threshold for iterations –

While running our experiments we have noticed that keeping the threshold at around 5000, results in almost no ties. We think this happens as a consequence of the graph being quite small and well connected, which results in either the Agent winning or losing in every run.

For this reason, we have maintained the threshold as 500 steps to implement all agents. This helps us avoid the case where the agent continuously moves away from the predator until the prey and agent are in the same cell.

The Complete Information Setting–

In this setting, the agent is aware of both the predator and prey position.

Agent 1 Implementation –

Initially the agent, predator and prey are spawned at a random location in the graph. We have made sure that the agent cannot initially spawn in either the predator or prey position.

```
index=list(range(1,no_of_nodes+1))
Agent_pos = np.random.choice(index)
index_pred=list(range(1,no_of_nodes+1))
index_pred.remove(Agent_pos)
predator_pos = np.random.choice(index_pred)
prey_pos = np.random.choice(index_pred)
```

The agent then calculates the shortest distance between its neighbors and the predator and prey. The distances are calculated using the Dijkstra's algorithm.

Based on these distances the agent decides which of its neighbors it must move to. The agent always tries to move towards the prey while moving away from the predator.
Once the agent moves, the prey and predator make their moves based on the rules they follow.

Termination Step -
If at any point the agent and prey are in the same node, the agent wins.
If at any point the predator and agent are in the same node, the predator wins.

Code -

```
def Agent1(adjacency_list):
    index=list(range(1,no_of_nodes+1))
    Agent_pos = np.random.choice(index)
    index_pred=list(range(1,no_of_nodes+1))
    index_pred.remove(Agent_pos)
    predator_pos = np.random.choice(index_pred)
    prey_pos = np.random.choice(index_pred)
    Agent1 = Agent(Agent_pos,0,0,0)
    Predator1 = Predator(predator_pos,0,0)
    Prey1 = Prey(prey_pos,0,0)
    count=0
    flag=0
    while(True):
        count=count+1
        if(count==threshold):
            break
        print("Agent1: ",Agent1.position," Predator: ",Predator1.position," Prey: ",Prey1.position)
        dist={}
        prey_list=[]
        for i in adjacency_list[Agent1.position]:
            temp_prey=djikstra(adjacency_list,i,Prey1.position)
            temp_pred=djikstra(adjacency_list,i,Predator1.position)
            dist[i]=[temp_prey,temp_pred]
        sorted_list_dist = sorted(dist.items(), key=lambda x:x[1])
        dist=dict(sorted_list_dist)
        print("dist ",dist)
        var=list(dist.keys())[0]
        agent_next_pos=0
        for i in dist:
            if(dist[i][0] == dist[var][0] and dist[i][1] >= 2):
                agent_next_pos=i

        maximus=0
        if(agent_next_pos==0):
            for i in dist:
                if(dist[i][1] != 0):
                    if(i>maximus):
                        agent_next_pos=i
                        maximus=i

        if(agent_next_pos!=0):
            Agent1.position=agent_next_pos
```

Prey movement –

```
l_prey=[]
for j in adjacency_list[Prey1.position]:
    l_prey.append(j)
l_prey.append(Prey1.position)
prey_next_point=np.random.choice(l_prey)
Prey1.position=prey_next_point
```

Predator Movement –

```
min_dist_agent=100000
for i in adjacency_list[Predator1.position]:
    temp=djikstra(adjacency_list,i,Agent1.position)
    print("Predator1 temp ",temp," i ", i)
    if(temp<min_dist_agent):
        pred_next_pos=i
        min_dist_agent=temp
Predator1.position=pred_next_pos
print("Predator1 new position ",Predator1.position)
```

Termination condition –

```
if(Agent1.position == Predator1.position):
    flag=2
    print("Predator Won")
    break
if(Agent1.position == Prey1.position):
    flag=1
    print("Agent won")
    break
```

Performance based on 3000 simulations (100 graphs run 30 times each)
How often the Predator catches the Agent –
12.867%
How often the Agent catches the Prey –
87.134%
How often the simulation hangs (no one has caught anything past a certain large time threshold) –
0 (Threshold – 500)

The Partial Prey Information Setting –

In this setting, the agent is always aware of the predator position. The agent also keeps track of a belief state which is necessarily a split of probabilities across all the graph nodes. In this setting, if the agent is not entirely sure where the prey is, it looks at its belief table and then assumes the prey to be at the node with the highest belief/probability at that time and then moves in accordance with the rules of agent 1.

Probability / Belief Calculations for Agent 3:

Here when the agent spawns in the graph at a node, the probability/ belief of the prey being there is zero (If prey was there, the game would've ended instantly). The probability table is a global variable in the code named as ***probability_table***.
The probability table indexes from 0 to 49 for nodes 1 to 50 respectively.
Now, before the survey, the probabilities are divided equally at each node for prey being there. So, after spawning the agent the probability at each node is 1/49 except the node where agent currently is (where it is zero).

Once we do a survey, we have two cases:

1. ***Prey is found in the survey:*** In this case, all the values are set to zero in the probability table, except the survey node where it is set to 1.
2. ***Prey is not found in the survey:*** In this case, except two places (Node where agent is currently present and the survey node) all other values in the probability table equal 1/48 (indicating equal belief) of prey being in any cell. The value at agent node and the survey node is zero.

This is the way we calculate probabilities after each survey.

The transition probabilities after the prey moves are updated in the probability table as follows:

PROBABILITY CALCULATION WHEN THE AGENT CAN'T SEE THE PREY (AGENT-3).

→ SUPPOSE AFTER THE SURVEY ON A RANDOM NODE (NODE = 20), WE DON'T FIND THE PREY.

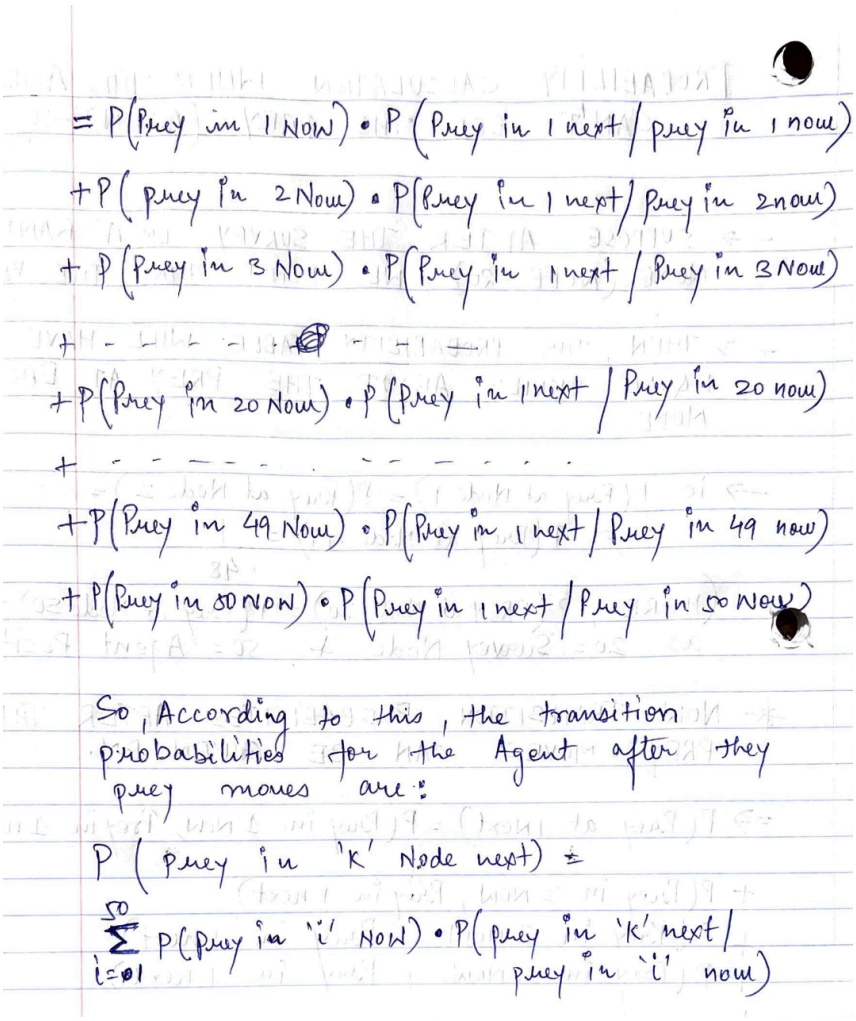
→ THEN, THE PROBABILITY TABLE WILL HAVE EQUAL BELIEF ABOUT THE PREY AT EACH NODE

→ ie. $P(\text{Prey at Node 1}) = P(\text{Prey at Node 2}) = \cdots = P(\text{Prey at Node 49}) = \frac{1}{48}$

(HERE, $P(\text{PREY at Node 20}) = P(\text{Prey at Node 50}) = 0$ as 20 = Survey Node & 50 = Agent Position.

* NOW TRANSITION PROBABILITES AFTER THE PREY MOVES CAN BE GIVEN BY.

$\Rightarrow P(\text{Prey at 1 Next}) = P(\text{Prey in 1 Now, Prey in 1 next})$
$+ P(\text{Prey in 2 NOW, Prey in 1 next})$
$+ P(\text{Prey in 3 NOW, Prey in 1 next})$
$+ P(\text{Prey in 4 NOW, Prey in 1 next})$
$+ \cdots$
$+ P(\text{Prey in 20 NOW, Prey in 1 Next})$
$+ \cdots P(\text{Prey in 49 Now, Prey in 1 next})$
$+ P(\text{Prey in 50 Now, Prey in 1 next}).$



This is how the transition probabilities are calculated. Here, the second term in the conditional probability depends on the neighbors of where the Prey is assumed to be. (In the "Prey in 'i' now term).

We have implemented the transition probabilities in the function ***probability_distribution2()***. Here, the argument is the adjacency list of the graph. The values in the ***probability_table*** are updated in this function.

Agent 3 Implementation –

In this, the agent spawns at a random location as usual and predator and prey also spawn at random nodes in the graph. We have made sure that the prey and predator do not spawn at the agent's location.

```
def Ag3(adjacency_list):
    index=list(range(1,no_of_nodes+1))
    Agent_pos = np.random.choice(index)
    index_pred=list(range(1,no_of_nodes+1))
    index_pred.remove(Agent_pos)
    predator_pos = np.random.choice(index_pred)
    prey_pos = np.random.choice(index_pred)

    Agent1 = Agent(Agent_pos,0,0,0)
    Predator1 = Predator(predator_pos,0,0)
    Prey1 = Prey(prey_pos,0,0)
```

After this, we have assigned each value in the probability table as 1/49 except the Agent node.

```
probability_table[Agent1.position-1]=0

for y in range(no_of_nodes):
    if(y != Agent1.position-1):
        probability_table[y]=(1/(no_of_nodes-1))
```

After this we pick a random node from the graph (except the agent node) for surveying and perform the survey using the Survey method. The Survey method returns 1 if the prey is found there and 0 if prey is not found in the survey node. In this method, we pass the survey node and agent position as arguments.

```
def Survey(index,prey):
    if(index==prey):
        return 1
    else:
        return 0
```

```
#Initial Survey
surveyList = list(range(1,no_of_nodes+1))
surveyList.remove(Agent_pos)
surveyNode = np.random.choice(surveyList)

surveyRes = Survey(surveyNode, Prey1.position)
```

If the survey result is 1, then I update the probability table as explained above.

```
    if(surveyRes == 1):
        #Prey Found -> Update Probab table
        for i in range(len(probability_table)):
            probability_table[i] = 0
        probability_table[surveyNode-1] = 1
```

After that we move the agent in accordance with rules of agent 1.

If the survey result is 0, then prey is not found in the survey, and I update the probability table as explained above.

```
    else:
        #Prey Not Found -> Update Probab table
        print("Agent Position", Agent1.position)
        for i in range(len(probability_table)):
            probability_table[i] = 1/(no_of_nodes-2)
        probability_table[surveyNode-1] = 0
        probability_table[Agent1.position-1] = 0
```

In this case, since we don't know the exact location of the prey, we move away from the predator as all other values in the probability table are equal.

In every timestep when the prey moves, we call the method to update the probability table with the transition probabilities. This is given by:

```
def probability_distribution2(graph):

    #Previous Probabilities
    copy_prob_table = [None] * len(probability_table)
    for i in range(len(probability_table)):
        copy_prob_table[i] = probability_table[i]

    #Calculating and populating next probabilities after the prey has moved
    for i in range(len(probability_table)):
        prob_res_i = 0
        for j in range(len(probability_table)):
            if((i+1) in graph[j+1] or (i+1) == (j+1)):
                fact2 = 1/(len(graph[j+1]) + 1)
            else:
                fact2 = 0
            prob_res_i += copy_prob_table[j] * fact2

        probability_table[i] = prob_res_i
```

After this at the end of the loop, we survey the node with the maximum probability of containing the prey.

```
# Take the max of the probability table for position of the new survey
maxProbab = max(probability_table)
maxPosition = probability_table.index(maxProbab) + 1 #Adding 1 to compensate for the indices

# Do a survey on this maxPosition
#prey_pos = Prey1.position
surveyNode = maxPosition
print("Survey Node:", surveyNode)
surveyRes = Survey(surveyNode, Prey1.position)
```

We do the above steps with a threshold 500 and terminate when the agent catches the prey or the predator catches the agent.
Performance based on 3000 simulations (100 graphs run 30 times each)

How often the Predator catches the Agent –
17.934%

How often the Agent catches the Prey –
77.534%

How often the simulation hangs (no one has caught anything past a certain large time threshold) –
136 (Threshold – 500)

How often the Agent knows exactly where the Prey is during a simulation where that information is partial –
4.8%

The Partial Predator Information Setting –

In this setting, the agent is always aware of the prey position. The agent also keeps track of a belief state which is necessarily a split of probabilities across all the graph nodes. In this setting, when the agent is not entirely sure where the predator is, it looks at its belief table and then assumes the predator to be at the node with the highest belief/probability at that time and then moves in accordance with the rules of agent 1.

Probability / Belief Calculations for Agent 5:

Here when the agent spawns in the graph at a node, the probability/ belief of the predator being there is zero (If predator was there, the game would've ended instantly). The probability table for predator also is a global variable in the code named as ***pred_probab_table***.

The probability table indexes from 0 to 49 for nodes 1 to 50 respectively.

Now, before the survey, the probabilities are divided equally at each node for predator being there. So, after spawning the agent the probability at each node is 1/49 except the node where agent is currently there (where it is zero).

Once we do a survey, we have two cases:

1. ***Predator is found in the survey****:* In this case, all the values are set to zero in the predator probability table, except the survey node where it is set to 1.
2. ***Predator is not found in the survey****:* In this case, except two places (Node where agent is currently present and the survey node), all other values in the predator probability table equal 1/48 (indicating equal belief). The value at agent node and the survey node is zero.

This is the way we calculate probabilities after each survey.

The formula used for the transition probabilities is –

P (Predator in 'kth' node next) = Summation(i -> 1 to 50) over all nodes(P(Predator in 'ith' node now) * P(Predator in 'kth' node next / Predator in 'ith' node now))

The transition probabilities after the predator moves are updated in the predator probability table as follows:

1. ***If the predator is in distracted state (60% chance):*** In this case, the predator behaves in a similar manner to the prey. It doesn't choose the neighbor with the shortest path to the predator rather it picks a neighbor to go to on

random. So, here we have used the same logic as the transition probability in case of the partial prey information. Only change here is that the predator can't stay in place, so there is one less move possible for the prey in this. This is implemented in the function *prob_distracted(graph).*

```
def prob_distracted(graph):
    #Previous Probabilities
    copy_prob_table = [None] * len(pred_probab_table)
    for i in range(len(pred_probab_table)):
        copy_prob_table[i] = pred_probab_table[i]

    #Calculating and populating next probabilities after the prey has moved

    for i in range(len(pred_probab_table)):
        prob_res_i = 0
        for j in range(len(pred_probab_table)):
            #print("J = ", j)
            if((i+1) in graph[j+1]):
                fact2 = 1/(len(graph[j+1]))
            else:
                fact2 = 0
            prob_res_i += copy_prob_table[j] * fact2

        pred_probab_table[i] = prob_res_i
```

2. ***If the predator is moving to close the distance to prey (40% chance):*** In this case the transition probabilities are calculated using the same formula but the neighbors which the predator can go to are more restricted. In this, we have calculated the shortest distance to the agent position from all neighbors of the predator. The moves available next to the predator is the neighbor with the shortest distance from the agent. If there are 2 neighbors with same shortest distance, then the probability is evenly divided between them. If from all three neighbors, the distance is same -> again probability is evenly divided between them. In the case of only one neighbor having shortest distance to agent, the probability of predator choosing that neighbor for next move is 1 and we know the position of the predator in the next move as well. This is implemented in the function ***probab_pred_distribution(graph,agentPos).***

```
def probab_pred_distribution(graph,agentPos):

    #Previous Probabilities
    copy_prob_table = [None] * len(pred_probab_table)
    for i in range(len(pred_probab_table)):
        copy_prob_table[i] = pred_probab_table[i]

    #Calculating and populating next probabilities after the prey has moved
    for i in range(len(copy_prob_table)):
        prob_res_i = 0
        for j in range(len(copy_prob_table)):
            if((i+1) in graph[j+1]):
                min_pred_dist = 0
                pred_Arr = []
                min_d = 100000
                min_d_i = []
                for x in graph[j+1]:
                    tempdist = djikstra(graph, x, agentPos)
                    tupEl = (x,tempdist)
                    pred_Arr.append(tupEl)
                for g in range(len(pred_Arr)):
                    if(pred_Arr[g][1] <= min_d):
                        min_d = pred_Arr[g][1]
                        min_d_i.append(pred_Arr[g][0])


                options = 0
                for g in pred_Arr:
                    if(min_d == g[1]):
                        options +=1
                if(options == 1):
                    co_or = min_d_i[0] - 1
                    for t in range(len(pred_probab_table)):
                        pred_probab_table[t] = 0
                        pred_probab_table[co_or] = 1

            elif(options == 2):
                co_or1 = min_d_i[0] - 1
                co_or2 = min_d_i[1] - 1
                for t in range(len(pred_probab_table)):
                    pred_probab_table[t] = 0
                pred_probab_table[co_or1] = 1/2
                pred_probab_table[co_or2] = 1/2
            elif(options == 3):
                co_or1 = min_d_i[0] - 1
                co_or2 = min_d_i[1] - 1
                co_or3 = min_d_i[2] - 1

                for t in range(len(pred_probab_table)):
                    pred_probab_table[t] = 0
                pred_probab_table[co_or1] = 1/3
                pred_probab_table[co_or2] = 1/3
                pred_probab_table[co_or3] = 1/3

            else:
                for t in range(len(pred_probab_table)):
                    pred_probab_table[t] = 0

        else:
            for t in range(len(pred_probab_table)):
                pred_probab_table[t] = copy_prob_table[t]
```

Agent 5 Implementation –

In this, the agent spawns at a random location as usual and predator and prey also spawn at random nodes in the graph. We have made sure that the prey doesn't spawn at the agent's location and the predator doesn't spawn at the agent's location.

```
def Ag5(adjacency_list):
    index=list(range(1,no_of_nodes+1))
    Agent_pos = np.random.choice(index)
    index_pred=list(range(1,no_of_nodes+1))
    index_pred.remove(Agent_pos)
    predator_pos = np.random.choice(index_pred)
    prey_pos = np.random.choice(index_pred)

    Agent1 = Agent(Agent_pos,0,0,0)
    Predator1 = Predator(predator_pos,0,0)
    Prey1 = Prey(prey_pos,0,0)
```

After this, we have assigned each value in predator probability table as 1/49 except the node of the agent.

```
    pred_probab_table[Agent1.position-1]=0

    for y in range(no_of_nodes):
        if(y != Agent1.position-1):
            pred_probab_table[y]=(1/(no_of_nodes-1))
```

Then we perform a survey on any random node in the graph except the agent node.

```
    #Initial Survey
    surveyList = list(range(1,no_of_nodes+1))
    surveyList.remove(Agent_pos)
    surveyNode = np.random.choice(surveyList)

    surveyRes = Survey(surveyNode, Predator1.position)
```

Now, we go into the while loop till we either cross the threshold number of steps set for the experiment or the agent catches the prey, or the agent is caught by the predator.

Then we update the probabilities as explained above based upon the survey result of 1 or 0. Here, 1 signifies that predator was found in the survey and 0 signifies that predator was not found in the survey.

If the predator is found in the survey, we move the agent towards the prey and away from the predator using the rules of Agent 1.

If the predator is not found in the survey, then the probabilities are equal everywhere of predator being in any node. So, in this case we move the agent towards the prey choosing the neighbor from where we get shortest distance to prey.

After this we move the prey to a random neighbor as specified.

```
#Now moving the prey
l_prey=[]
for j in adjacency_list[Prey1.position]:
    l_prey.append(j)
l_prey.append(Prey1.position)
prey_next_point=np.random.choice(l_prey)
Prey1.position=prey_next_point
```

Now when the predator has to move, it choses 40% of the times to move to the neighbor with the shortest distance from the prey and 60% of the times it is distracted, in which it chooses a neighbor to move to at random. In both cases probability tables are updated by calling functions ***probab_pred_distribution*** and ***prob_distracted*** respectively as explained above.

```
#Move predator
pred_choice=np.random.choice([0,1],p=[0.6,0.4])
if(pred_choice==1):
    min_dist_agent=100000
    for i in adjacency_list[Predator1.position]:
        temp=djikstra(adjacency_list,i,Agent1.position)
        print("Predator1 temp ",temp," i ", i)
        if(temp<min_dist_agent):
            pred_next_pos=i
            min_dist_agent=temp
    Predator1.position=pred_next_pos
    #Update belief Probablity Table of Agent
    probab_pred_distribution(adjacency_list, Agent1.position)
elif(pred_choice==0):
    Predator1.position=np.random.choice(adjacency_list[Predator1.position])
    prob_distracted(adjacency_list)
```

We then again do a survey of the node with the highest probability of containing the predator.

```
# Take the max of the probability table for position of the new survey
maxProbab = max(pred_probab_table)
maxPosition = pred_probab_table.index(maxProbab) + 1 #Adding 1 to compensate for the indices

# Do a survey on this maxPosition
#prey_pos = Prey1.position
surveyNode = maxPosition
print("Survey Node:", surveyNode)
surveyRes = Survey(surveyNode, Predator1.position)
```

We then go back in the loop and rerun the logic explained above till we reach one of the termination conditions.

Performance based on 3000 simulations (100 graphs run 30 times each)

How often the Predator catches the Agent –
26.534%

How often the Agent catches the Prey –
73.467%

How often the simulation hangs (no one has caught anything past a certain large time threshold) –
0 (Threshold – 500)

How often the Agent knows exactly where the Predator is during a simulation where that information is partial –
62.52%

The Combined Partial Information Setting –

In this setting the agent does not always know the exact location of the prey as well as the predator. The agent also keeps track of a belief state for both the prey and predator which is necessarily a split of probabilities across all the graph nodes. The probabilities/ beliefs are propagated for each node in the graph as the agent moves and does surveys.

Here, the agent begins by knowing the location of the predator. So, initially we do a survey for the prey. The predator surveys have been given preference in the writeup. Only when we know the exact position of the predator and not know the exact location of the prey, we go for a prey survey.

The probabilities/beliefs are maintained in a similar way for Agent 7 like we have done for Agent 3 and Agent 5.

Here, we have used 2 flag variables - flag_preySurvey, flag_predSurvey. These are set (value = 1) when we know the exact position of prey and predator respectively.

There are 4 major conditions in the code:

1. ***When Agent knows about predator location and the survey result for prey is 1***: In this case we update the probability table/belief corresponding to the prey such that only the value corresponding to the survey node is 1 in the probability table. Then we move the Agent with rules from Agent 1 as now we have both Predator and Prey location. After the prey moves, we call the function ***probability_distribution2*** as in agent 3 to update the probability/belief table with transition probabilities. Then the predator moves. We use the similar logic as in agent 5 for choosing and implementing the transition probabilities logic with the distracted predator (60% of the time) and shortest distance to the agent (40% of the time).

2. ***When Agent knows about the predator location and survey result for prey is 0:*** In this case we update the probability table of prey with 2 nodes as 0(Agent node and the survey node). All other nodes are given equal belief (1/48) in this case for containing the prey. After this, since all the probabilities are equal for the neighbor nodes for prey, we emphasize on moving away from the predator as that location is known definitively to us. Now the prey moves, and we update the probability table with transition probabilities for prey similar as above using the ***probability_distribution2*** method. After this, the predator moves, and we use the same logic as above to choose between distracted predator and shortest distance to prey. We update the transition probabilities again in each case with methods ***prob_distracted*** and ***prob_pred_distribution*** respectively.

3. ***When Agent knows about prey location and survey result for predator is 1:*** In this case we update the probability table for predator with 1 at the survey location and rest all with zero. Then we move the agent in accordance with rules from agent 1 as we know the exact position of the predator and prey. Then we move the prey and update the corresponding transition probability of the prey using ***probability_distribution2*** method. After this, the predator moves, and the transition probabilities are updated as in the above cases.

4. ***When Agent knows about predator location and survey result for predator is 0:*** In this case, we update the probability table for predator with 0 in agent location and survey location and update the rest of the table with equal split of probabilities (1/48). After this as the location of predator is unknown and the beliefs are evenly distributed for the predator location we emphasize on moving towards the prey as the prey location is definitively known. Then we

move the prey and update the transition probabilities as above steps. Then we move the predator and update the transition probabilities as above cases.

5. ***When Agent doesn't know about predator location and prey location:*** In this case the agent doesn't know definitively about the predator or prey location. In this case we execute a predator survey. In the movement of the agent, we look at the probability/belief tables of predator and prey and take the highest values from them and assume the predator and prey to be in those indexes. Then we calculate the distances from agent's neighbors to the prey and predator locations. After this the agent moves in accordance with rules from agent 1. Then we update the probability tables for prey and predator after their respective moves.

   Once these steps are executed, we check if the predator location is known. This is calculated by a check if 1 exists in the predator belief table. If we know the location of the predator, then we do a survey in accordance with agent 3 to survey for the prey location.

```
# Checking for next survey
if( 1 in pred_probab_table ): # Agent knows where pred is -> Survey for prey in next
    flag_preySurvey = 0
    flag_predSurvey = 1

    # Take the max of the probability table for position of the new survey
    maxProbab = max(probability_table)
    maxPosition = probability_table.index(maxProbab) + 1 #Adding 1 to compensate for the indices

    # Do a survey on this maxPosition
    #prey_pos = Prey1.position
    surveyNode = maxPosition
    print("Survey Node:", surveyNode)
    surveyRes = Survey(surveyNode, Prey1.position)
```

   If predator location is not definitively known, then we go for a survey for the predator position as is given in the writeup.

```
else: # Agent doesn't know where pred is -> Survey for pred

    flag_predSurvey = 0
    flag_preySurvey = 1

     # Take the max of the probability table for position of the new survey
    maxProbab = max(pred_probab_table)
    maxPosition = pred_probab_table.index(maxProbab) + 1 #Adding 1 to compensate for the indices

    # Do a survey on this maxPosition
    #prey_pos = Prey1.position
    surveyNode = maxPosition
    print("Survey Node:", surveyNode)
    surveyRes = Survey(surveyNode, Predator1.position)
```

We keep doing the above steps till we reach our any of our termination conditions – the predator catches the agent, the agent catches the prey or the threshold number of steps are reached.

Performance based on 3000 simulations (100 graphs run 30 times each)

How often the Predator catches the Agent –
26.234%

How often the Agent catches the Prey –
73.757%

How often the simulation hangs (no one has caught anything past a certain large time threshold) –
0 (Threshold – 500)

How often the Agent knows exactly where the Predator is during a simulation where that information is partial –
60.323%

How often the Agent knows exactly where the Prey is during a simulation where that information is partial –
2.634%

Even agent Strategy –

Including all the conditions and tiebreakers for the agents, we have implemented one additional strategy for the even numbered agents.
The even numbered agents decide whether to pursue the prey or move away from the predator based on the proximity of both the prey and predator.

Initially the agent calculates the distance from its neighbors to the predator and prey. If the distance to the predator from a certain neighbor is less than 5, the neighbor and its distance is inserted into a dictionary called ‘dist_pred’. Similarly, if the distance to the prey from a certain neighbor is less than 5, the neighbor and its distance is inserted into a dictionary called ‘dist_prey’.

If both dist_pred and dist_prey remain empty after iterating over all the agent's neighbors, then the agent 1's strategy of tie breaking is implemented between the neighbors of the agent.

In case dist_pred/dist_prey is not empty, then the agent prioritizes moving away from the predator over moving towards the prey.

```
  dist_prey={}
  dist_pred={}
  dist={}
  for i in adjacency_list[Agent1.position]:
      temp_prey=djikstra(adjacency_list,i,Prey1.position)
      temp_pred=djikstra(adjacency_list,i,Predator1.position)
      if(temp_prey<5):
          dist_prey[i]=[temp_prey]
      if(temp_pred<5):
          dist_pred[i]=[temp_pred]
      if(temp_prey>=5 and temp_pred>=5):
          dist[i]=[temp_prey,temp_pred]
  if(len(dist_pred)>0):
      sorted_list_dist = sorted(dist_pred.items(), key=lambda x:x[1])
      dist_pred=dict(sorted_list_dist)
      print("dist_pred ",dist_pred)
      agent_next_pos=list(dist_pred.keys())[len(dist_pred)-1]
  elif(len(dist_prey)>0):
      sorted_list_dist = sorted(dist_prey.items(), key=lambda x:x[1])
      dist_prey=dict(sorted_list_dist)
      print("dist_prey ",dist_prey)
      agent_next_pos=list(dist_prey.keys())[0]
else:
    sorted_list_dist = sorted(dist.items(), key=lambda x:x[1])
    dist=dict(sorted_list_dist)
    print("dist ",dist)
    var=list(dist.keys())[0]
    agent_next_pos=0
    for i in dist:
        if(dist[i][0] == dist[var][0] and dist[i][1] >2):
            agent_next_pos=i

    maximus=0
    if(agent_next_pos==0):
        for i in dist:
            if(dist[i][1] != 0):
                if(i>maximus):
                    agent_next_pos=i
                    maximus=i

if(agent_next_pos!=0):
    Agent1.position=agent_next_pos
print("Agent new position ",Agent1.position)
```

As shown above, in case both dist_pred and dist_prey remain empty, the agent moves to the neighbor based on the rules followed by agent 1.

Agent 2 –

How often the Predator catches the Agent –
1.634%
How often the Agent catches the Prey –
93.4%
How often the simulation hangs (no one has caught anything past a certain large time threshold) –
149 (Threshold – 500)

Agent 4 –

How often the Predator catches the Agent –
6.967%
How often the Agent catches the Prey –
87.967%
How often the simulation hangs (no one has caught anything past a certain large time threshold) –
152

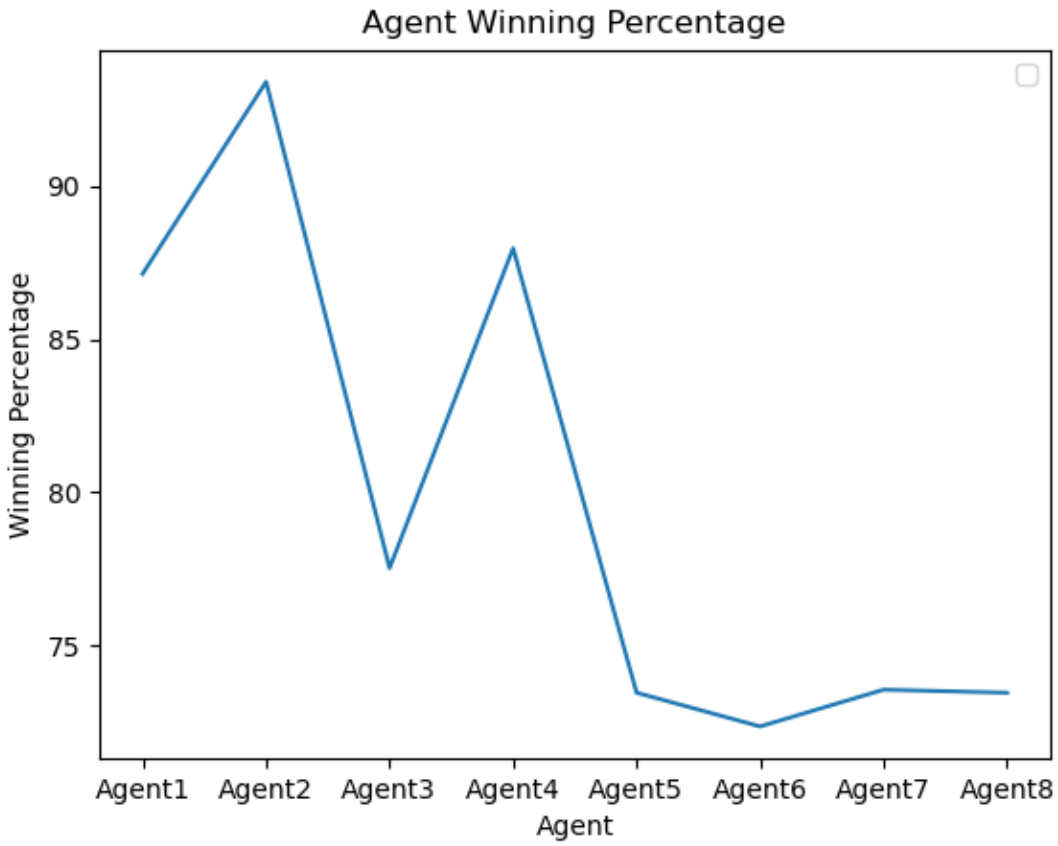
Agent 6 –

How often the Predator catches the Agent –
27.634%
How often the Agent catches the Prey –
72.367%
How often the simulation hangs (no one has caught anything past a certain large time threshold) –
0

Agent 8 –

How often the Predator catches the Agent –
26.534%

How often the Agent catches the Prey –
73.4667%
How often the simulation hangs (no one has caught anything past a certain large time threshold) –
0



FAULTY DRONE –

We have created a new function to implement the faulty survey condition. Here, we have 2 options (0 and 1), out of which 0 is chosen 90% of the times and 1 is chosen 10% of the times. If the survey node and the predator/prey position are same and the selected option is 0, we return 1 as the result from this function which signifies the prey to be there. Otherwise, we return 0 indicating that the prey is not there. In this case, the faulty condition is realized as even though the prey/predator is at the node the function will return 0 signifying that prey/predator is not present at that node.

CODE FOR FAULTY DRONE :

```
def fault_survey(index,location):

    fault_surv = np.random.choice([0,1],p=[0.9,0.1])

    if(index == location and fault_survey == 0):
        return 1
    else:
        return 0
```

AGENT 7 With Faulty Drone Results:

How often the Predator catches the Agent –
26.567%

How often the Agent catches the Prey –
73.433%

How often the simulation hangs (no one has caught anything past a certain large time threshold) –
0 (Threshold – 500)

How often the Agent knows exactly where the Predator is during a simulation where that information is partial –
45.323%

How often the Agent knows exactly where the Prey is during a simulation where that information is partial –
0.634%

AGENT 8 With Faulty Drone Results:

How often the Predator catches the Agent –
27.645%

How often the Agent catches the Prey –
72.355%

How often the simulation hangs (no one has caught anything past a certain large time threshold) –
0 (Threshold – 500)

How often the Agent knows exactly where the Predator is during a simulation where that information is partial –
43.323%

How often the Agent knows exactly where the Prey is during a simulation where that information is partial –
0.741%