Artificial Intelligence Project 3 (CS520)

Better, Smarter, Faster

Viswajith Menon (VM623)

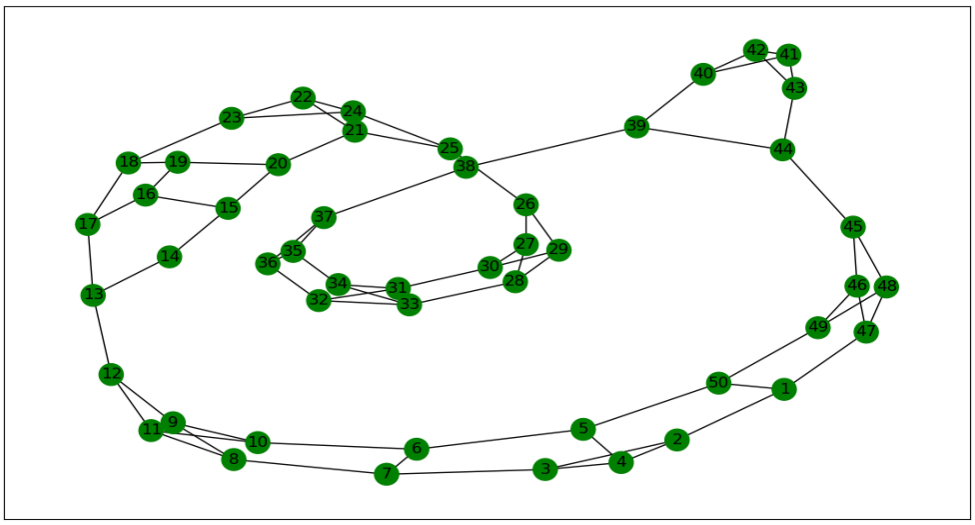
Zeeshan Ahsan (ZA224)

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.

Sample environment –

Adjacency list - {1: [50, 2, 47], 2: [1, 3, 4], 3: [2, 4, 7], 4: [3, 5, 2], 5: [4, 6, 50], 6: [5, 7, 10], 7: [6, 8, 3], 8: [7, 9, 11], 9: [8, 10, 12], 10: [9, 11, 6], 11: [10, 12, 8], 12: [11, 13, 9], 13: [12, 14, 17], 14: [13, 15], 15: [14, 16, 20], 16: [15, 17, 19], 17: [16, 18, 13], 18: [17, 19, 23], 19: [18, 20, 16], 20: [19, 21, 15], 21: [20, 22, 25], 22: [21, 23, 24], 23: [22, 24, 18], 24: [23, 25, 22], 25: [24, 26, 21], 26: [25, 27, 29], 27: [26, 28, 30], 28: [27, 29, 33], 29: [28, 30, 26], 30: [29, 31, 27], 31: [30, 32, 34], 32: [31, 33, 36], 33: [32, 34, 28], 34: [33, 35, 31], 35: [34, 36, 37], 36: [35, 37, 32], 37: [36, 38, 35], 38: [37, 39], 39: [38, 40, 44], 40: [39, 41, 42], 41: [40, 42, 43], 42: [41, 43, 40], 43: [42, 44, 41], 44: [43, 45, 39], 45: [44, 46, 48], 46: [45, 47, 49], 47: [46, 48, 1], 48: [47, 49, 45], 49: [48, 50, 46], 50: [49, 1, 5]}



State Space–

The collection of all possible configurations for a particular environment is known as a state space. Each configuration is called a state.

For this project we have defined a state at a particular instant as a list of 3 elements

[Agent Position, Prey Position, Predator Position]

```
curState = [Agent1.position,Prey1.position,Predator1.position]
```

How many distinct states (configurations of predator, agent, prey) are possible in this environment?

This environment contains states resembling every combination of positions the agent, prey and predator can occupy. As there are 50 nodes in the graph (environment), each player can have a value from 1-50 which results to a total of – 125,000 states.

Utility–

The utility of a state gives us a quantitative value to determine the quality of a state. The utility of a state helps the agent determine whether moving to this new state is beneficial to its ultimate goal of catching the prey while evading the predator.

For this environment U* of a state represents the minimum number of steps taken by the agent to catch the prey.

What states s are easy to determine $U*$ for?

- States where the prey and agent are in the same position and the predator is away from them –
  These states represent a winning position for the agent and the U* for these states can be determined as 0.
  For example - [23,23,32], [5,5,15], [45,45,13] etc

- States where the predator and agent are in the same position and the prey is away from them –
  These states represent a losing position for the agent as the predator has caught the agent. The U* for these states can be defined as infinity.
  For example – [23,32,23], [5,15,5], [45,13,45] etc

- States where the predator, prey and agent are in the same position –
  These states again represent a losing position for the agent as the predator cannot kill the prey which is present in the same cell as the predator. The U* for these states can be defined as infinity.
  For example – [23,23,23], [5,5,5] etc

- States where the prey is 1 step away from the agent –
  U* for these states are 1 as the agent will always only need 1 move/round to kill the prey.

Utility Implementation for complete info setting–

Our implementation aims to assign utilities in a way that more favorable positions have a lower numerical value of utility. This allows the agent to check it's neighbor's utility values and make a decision based on which neighbor has the lowest utility. Utilities are stored using a dictionary, where the 'key' represents a state and the 'value' represents the utility of that state.

Utility Initialization –
We have initialized the utilities in a way that reflects our final goal for this project.

- States that represent a predator win are initialized with a value of infinity.
- States which represent an agent win is defined with a utility of 0.
- States which represent a configuration where the agent and prey are 1 step away are defined with a utility of 1.
- States which represent a configuration where the agent and predator are 1 step away are defined with a utility higher than every other state in the state space (except the terminal states). For these states, the value assigned at initialization is 10.
- Every other state in the state space is initialized with a value of 7.

```
def utility_initializer(adjacency_list):
    for agent in range(1,no_of_nodes+1):
        for prey in range(1,no_of_nodes+1):
            for pred in range(1,no_of_nodes+1):
                if(agent==pred):
                    utility[agent,prey,pred] = math.inf
                else:
                    if(agent==prey):
                        utility[agent,prey,pred] = 0
                    elif(djikstra(adjacency_list,agent,prey)==1):
                        utility[agent,prey,pred] = 1
                    elif(djikstra(adjacency_list,agent,pred)==1):
                        utility[agent,prey,pred] = 10
                    else:
                        utility[agent,prey,pred] = 7
```

Reward Initialization –

A reward dictionary is maintained to represent a reward which is defined for every state. For our implementation, the reward resembles a cost the agent pays to transition to a new state.

The reward setup for minimizing utility for favorable positions –

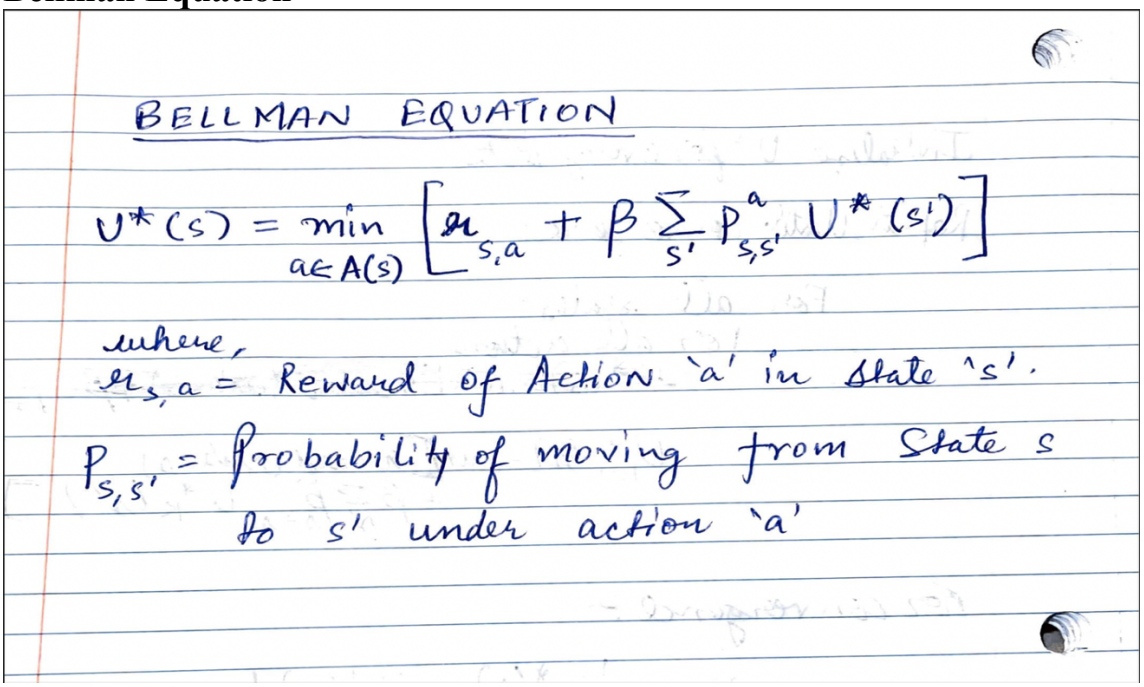
- States where the agent and predator share positions, reward = 1
- States where the agent and prey share positions, reward = -1
- Reward for every other state = 0

```
def reward_vector():
    for agent in range(1,no_of_nodes+1):
        for prey in range(1,no_of_nodes+1):
            for pred in range(1,no_of_nodes+1):
                if(agent==pred):
                    reward[agent,prey,pred] = 1
                if(agent==prey and agent!=pred):
                    reward[agent,prey,pred] = -1
                else:
                    reward[agent,prey,pred] = 0
```

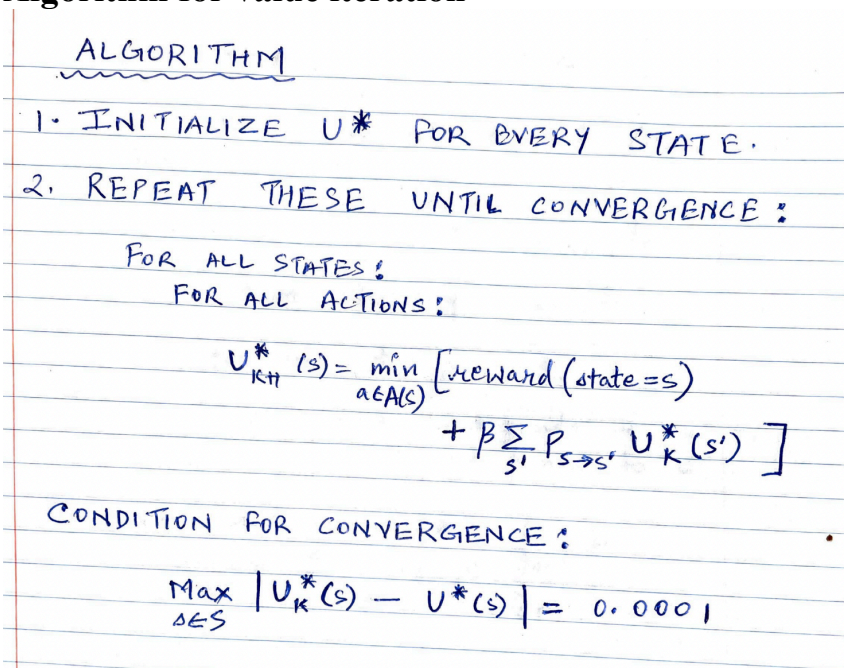
Value Iteration –

The Value Iteration algorithm helps us compute the optimal U* value for all the states by iteratively improving the initial estimate of U*(s) we had initialized.
The optimal values of U* are evaluated by running the Bellman equations until convergence.

Bellman Equation –



Algorithm for value iteration –



An example of how Value iteration evaluates the U* of a state in 1 iteration –

LET'S ASSUME THE AGENT IS AT POSITION 1 AND 1 HAS NEIGHBORS (1 → 2, 50, 5) &
SIMILARLY PREY IS AT 31 WITH NEIGHBORS (31 → 30, 32, 26) &
PREDATOR IS AT 40 WITH NEIGHBORS (40 → 41, 39, 37)

So, the current STATE OF the SYSTEM IS:
[1, 31, 40]

Now 3 ACTIONS POSSIBLE HERE ARE
a_1: 1 → 2 (Agent Moves from 1 to 2)
a_2: 1 → 50 (Agent Moves from 1 to 50).
a_3: 1 → 5 (Agent Moves from 1 to 5).

Then the states from these Actions can only be:

[1, 31, 40]

a_1 | a_2 | a_3

a_1	a_2	a_3
[2, 30, 41]	[50, 30, 41]	[5, 30, 41]
[2, 30, 39]	[50, 30, 39]	[5, 30, 39]
[2, 30, 37]	[50, 30, 37]	[5, 30, 37]
[2, 32, 41]	[50, 32, 41]	[5, 32, 41]
[2, 32, 39]	[50, 32, 39]	[5, 32, 37]
[2, 32, 37]	[50, 32, 37]	[5, 32, 37]
[2, 26, 41]	[50, 26, 41]	[5, 26, 41]
[2, 26, 39]	[50, 26, 39]	[5, 26, 39]
[2, 26, 37]	[50, 26, 37]	[5, 26, 37]

Thus, utility value of state [1,31,40] can be calculated as:

$$U_{(1,31,40)} = \min \Big\{ \Big(R_{(1,31,40)} + \beta \big(P_{(1,31,40)\to(2,30,41)} \cdot U_{(2,30,41)} + P_{(1,31,40)\to(2,30,39)} U_{(2,30,39)} + P_{(1,31,40)\to(2,30,37)} U_{(2,30,37)} + \cdots + P_{(1,31,40)\to(2,26,37)} V_{(2,26,37)} \big) \Big),$$

$$\Big(R_{(1,31,40)} + \beta \big(P_{(1,31,40)\to(50,30,41)} \cdot U_{(50,30,41)} + \cdots + P_{(1,31,40)\to(50,30,39)} \cdot V_{(50,30,39)} + P_{(1,31,40)\to(50,30,37)} \cdot U_{(50,30,37)} + \cdots + P_{(1,31,40)\to(50,26,37)} \cdot V_{(50,26,37)} \big) \Big),$$

$$\Big(R_{(1,31,40)} + \beta \big(P_{(1,31,40)\to(5,30,41)} \cdot U_{(5,30,41)} + \cdots + P_{(1,31,40)\to(5,30,39)} \cdot V_{(5,30,39)} + \cdots + P_{(1,31,40)\to(5,26,37)} \cdot V_{(5,26,37)} \big) \Big) \Big\}$$

How does U*(s) relate to U* of other states, and the actions the agent can take?

As explained above, for our environment the U* of a particular state is evaluated by running the value iteration algorithm to convergence. The number of actions an agent can make depends on the number of nodes the agent can choose to move to. In each iteration the U* of a particular state s is the minimum of the sum between the reward of the state s and the summation of the product of the probability of taking an action to move to s' from s and the utility of the state s'.

In our experiments we have observed that the optimal U* for any state evaluated after running value iteration to convergence closely resembles the distance between the agent and prey in that state.

Calculation of Transition probabilities between states –
We pre-compute the transition probability of all state changes and store it in a dictionary called 'trans_probab'.
The 'Key' in this dictionary are the 2 states we are calculating the transition probability for, and the 'value' is the probability.

In this we have to calculate the probability of the state change from S1 = S(p1,p2,p3) to S2= S(p4,p5,p6).

Here, the tuples contain (Agent Position, Prey Position and Prey Position) in order.

The probability that the system goes from state S1 to S2 is determined by the individual probabilities of the three elements in the tuple. This is justified as the agent moving from one position to another position is an independent event from the prey movement and predator given a particular state. The same reasoning translates to the prey and predator movement as they are also independent events in this system.

So, the transition probability can essentially be given as the product of individual probabilities of P (Agent going from p1 to p2), P (Prey going from p2 to p4) and P (Predator going from p3 to p6).

Thus, Transition Probability of the system going from State S1 -> State S2 = P (Agent going from p1 to p2) * P (Prey going from p2 to p4) * P (Predator going from p3 to p6).

Now, as the agent moves deterministically -> P (Agent going from p1 to p2) is always 1.
From the utility dictionary, we build the neighbors of each of Agent Position, Prey Position and Predator Position.

```
def transition_probab(adjacency_list):
    for key,val in utility.items():
        [agent,prey,pred] = key

        agent_nbrs = []
        for x in adjacency_list[agent] :
            agent_nbrs.append(x)

        prey_nbrs = []
        for x in adjacency_list[prey] :
            prey_nbrs.append(x)

        pred_nbrs = []
        for x in adjacency_list[pred] :
            pred_nbrs.append(x)
```

Now, we traverse the pred_nbrs for each Predator position and find out the shortest distance neighbor from that particular predator position. We put the neighbors with the shortest distance into a list(*shortest_dist_nbr_pred*). This is done to handle when we have more than one neighbor with shortest distance.

```
    shortest_dist_nbr_pred = []
    short_dist = 9999999
    for n in pred_nbrs:
        temp_pred_dist = djikstra(adjacency_list,n,agent)
        if(temp_pred_dist <= short_dist):
            short_dist = temp_pred_dist
            shortest_dist_nbr_pred.append(n)
```

After doing this we iterate through each Agent neighbor, Prey Neighbor and Predator Neighbor.

In this iteration, we check if the predator neighbor is in the *shortest_dist_nbr_pred* list. If it is present in the list then then the probability of going to that neighbor is (0.6 + 0.4*(1/len(pred_nbrs))) * 1/(len(*shortest_dist_nbr_pred)*.

This is justified as 60 % of the times the predator choses to go to its shortest distance to agent neighbor (thus 0.6), apart from this ; this neighbor can also be chosen randomly when predator is in the distracted state 40 % of the times and the probability of that happening is 1/length of list of predator neighbors(thus

0.4*(1/len(pred_nbrs)). Apart from this, we multiply this by 1/(len(shortest_dist_nbr_pred) to handle the condition when there is more than one shortest distance neighbor.

If the predator neighbor is not in the *shortest_dist_nbr_pred* list, that means the predator will move here only when it is in the distracted state(40 %) of the times and the probability of that neighbor being chosen is 1/length of the list of Predator Neighbors.
(thus 0.4*(1/len(*pred_nbrs*))

```
for i in agent_nbrs:
    for j in prey_nbrs:
        for k in pred_nbrs:
            if(k in shortest_dist_nbr_pred):
                pr = 1/(len(shortest_dist_nbr_pred))*(0.6 + (0.4*(1/len(pred_nbrs))))
            else:
                pr = 0.4 * (1/len(pred_nbrs))
```

For Prey movement, the probability calculation is straight forward as it moves randomly between its neighbors or chooses to stay in its current position with equal probability. So, this can be just given by *(1/(len(prey_nbrs) + 1))*.

Thus, the final transition probability can be given by:

```
trans_probab[(agent,prey,pred),(i,j,k)] = 1 * (1/(len(prey_nbrs) + 1)) * (pr)
```

Implementation of U* -

As described above the utilities and reward vector is first initialized.
Then the values are plugged into the Bellman equation and run until convergence,

```
def utility_star_1(adjacency_list):
    reward_vector()
    count = 0
    while(True):
        copy_utility={}
        for agent in range(1,no_of_nodes+1):
            for prey in range(1,no_of_nodes+1):
                for pred in range(1,no_of_nodes+1):
                    copy_utility[agent,prey,pred]=utility[agent,prey,pred]
        for agent in range(1,no_of_nodes+1):
            nbrs = []
            for x in adjacency_list[agent]:
                nbrs.append(x)
            for prey in range(1,no_of_nodes+1):
                for pred in range(1,no_of_nodes+1):
                    if(agent==prey):
                        continue
                    if(agent==pred):
                        continue
                    else:
                        min_val = math.inf
                        for x in nbrs:
                            summation = 0
                            cur_utility = 0
                            for temp_prey in adjacency_list[prey]:
                                for temp_pred in adjacency_list[pred]:
                                    transition_probability = trans_probab[(agent,prey,pred),(x,temp_prey,temp_pred)]
                                    summation = summation + (transition_probability * copy_utility[x,temp_prey,temp_pred])
                            cur_utility = reward[agent,prey,pred] + (1*summation)
                            if(cur_utility<min_val):
                                min_val=cur_utility
                        utility[agent,prey,pred] = min_val
                        print("utility[",agent,prey,pred,"]",utility[agent,prey,pred])
        flag=0
        for agent in range(1,no_of_nodes+1):
            for prey in range(1,no_of_nodes+1):
                for pred in range(1,no_of_nodes+1):
                    if(abs(utility[agent,prey,pred]-copy_utility[agent,prey,pred])>0.0001):
                        flag=1
                        break
        if(flag==0):
            break
```

Are there any starting states for which the agent will not be able to capture the prey? What causes this failure?

The states for which the agent will not be able to catch the prey –

- The states where the agent and predator share the same location.
  For example, [23,32,23], [41,5,41]
- The states where the agent, prey and predator share the same location.
  For example, [4,4,4], [18,18,18]
- The states where the predator and agent share a neighbor and that's the neighbor which has the lowest utility.

Find the state with the largest possible finite value of U* and give a visualization of it.

In our experiments, we have noticed that state configurations where the agent and predator are close together while the prey is away from them both have a higher value of U* compared to states where the distance between the agent and prey are less.

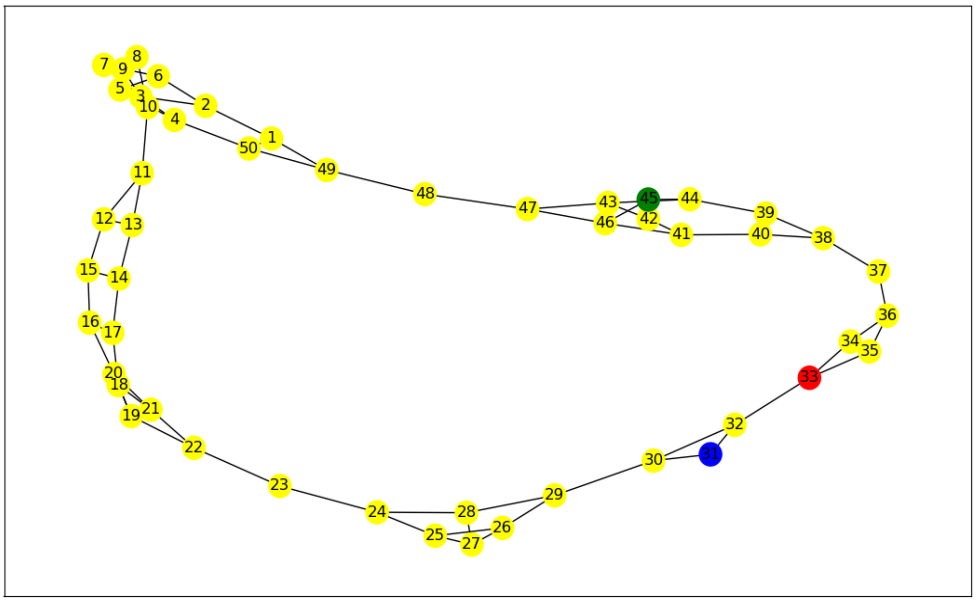
Example for 2 graphs –

Player representation –

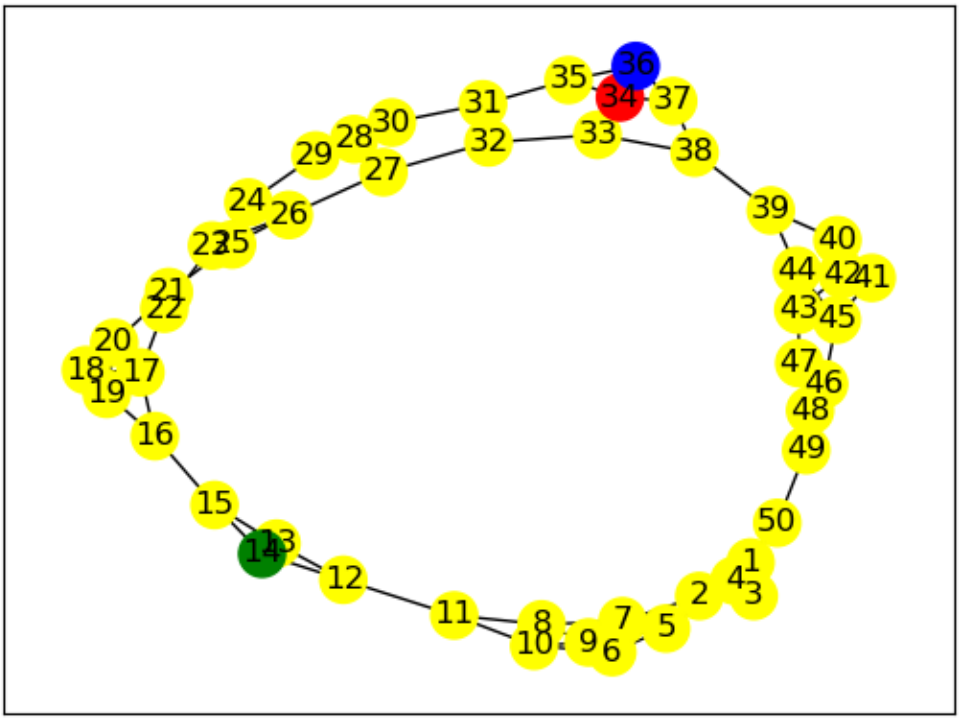
- Blue – Agent
- Green – Prey
- Red – Predator

Both these examples show states where the agent and predator are close together while the prey is away from them

State – [31,45,33]



State – [36,14,34]



U* Agent Implementation –

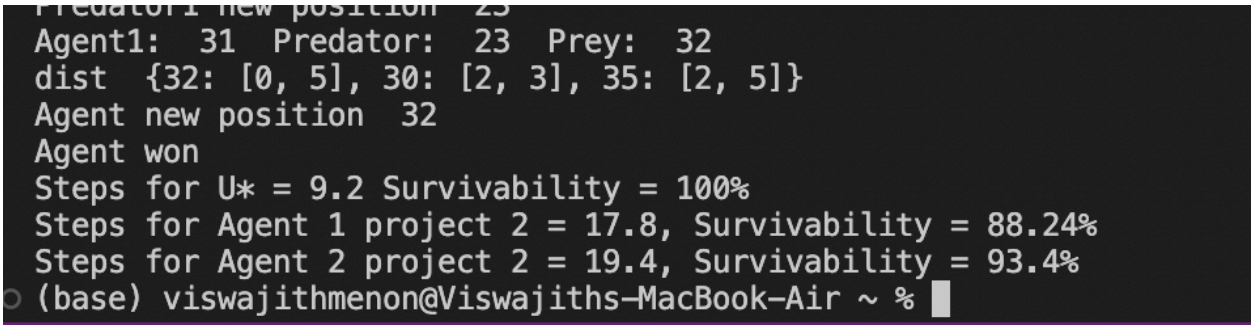
The agent implementation is very similar to the agent in project 2. The prey and predator move according to the same rules which governed them in project 2.

The only difference is that the agent moves to one of it's neighbors based on their utility values. The agent always moves to the neighbor which has the lowest utility value. If multiple neighbors have the same utility, it moves randomly to one of them.

```
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)

    curState = [Agent1.position,Prey1.position,Predator1.position]

    beta = 1
    count=0
    flag=0
    while(True):
        count = count+1
        if(count==500):
            break
        nbrs = []
        for x in adjacency_list[Agent1.position]:
            nbrs.append(x)

        min_nbr = 10000
        min_ut = 990
        utility_set=set()
        for x in nbrs:
            utility_set.add(utility[x,Prey1.position,Predator1.position])
            if(utility[x,Prey1.position,Predator1.position] < min_ut):
                min_ut = utility[x,Prey1.position,Predator1.position]
                min_nbr = x
        Agent1.position = min_nbr
        if(Agent1.position == Prey1.position):
            flag=1
            print("Agent won")
            break
        #Check if Predator Won
        if(Agent1.position == Predator1.position):
            flag=2
            print("Predator won")
            break

        # 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

        if(Agent1.position == Prey1.position):
            flag=1
            print("Agent won")
            break

        #Move predator
        pred_choice=np.random.choice([1,0],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)
                if(temp<min_dist_agent):
                    pred_next_pos=i
                    min_dist_agent=temp
            Predator1.position=pred_next_pos
        elif(pred_choice==0):
            Predator1.position=np.random.choice(adjacency_list[Predator1.position])

        #Check if Predator Won
        if(Agent1.position == Predator1.position):
            flag=2
            print("Predator won")
            break
```

Simulate the performance of an agent based on U* and compare its performance (in terms of steps to capture the prey) to Agent 1 and Agent 2 in Project 2. How do they compare?

The number of steps given below is the average number of steps taken by the agent to catch the prey in 3000 runs on one graph.



U* Agent –

- Survivability – 100%
- Number of steps taken – 9.2

Agent 1 from project 2 –

- Survivability – 88.24%
- Number of steps taken – 17.8

Agent 2 from project 2 –

- Survivability – 93.4%
- Number of steps taken – 19.4

As shows above it is clear that the U* agent outperforms the other 2 agents from project 2 in terms of step size and survivability. The reason the U* agent takes the least number of steps is because it now makes its decisions based on the utility of it's neighbors. The utility gives a good measure of the state of the agent's neighbor in terms of both the predator and prey locations and takes into account the future reward it might receive on making a particular move. The agent 1 and 2 from project 2 makes it's decision solely based on the distances from the prey and predator.

Agent 2 takes the highest number of steps as it prioritizes evading the predator more than moving towards the prey when the predator is in close proximity to the agent.

Are there states where the U* agent and Agent 1 make different choices? The U* agent and Agent 2? Visualize such a state, if one exists, and explain why the U* agent makes its choice.

As described above, the utility of a state takes future rewards into consideration when making a move, while the agent 1 and 2 solely move based on the current distance to the prey and predator. We have observed that whenever the closest path to the prey has a predator in between (in close proximity to the agent), the agent 1 and 2 will opt to evade the predator while the U* agent might see a future reward even though it is moving in a path considerably making it closer to the predator.

Model V for complete information setting –

We have stored the utility values as a dictionary with key as [Agent Position, Prey Position, Predator Position].

How do you represent the states 's' as input for your model? What kind of features might be relevant?

For the model, we extract the Agent Position, Prey position and Predator position from the state and use them as columns/features in our model. Apart from them, we have also used the distance between agent-prey and agent-predator as features for our model. These are the relevant features from the graph for our problem and we extract this from the graph after running Djikstra algorithm for these. So, we have features for our Dataset which are:

A. Agent Position
B. Prey Position
C. Predator Position
D. Distance between Agent and Prey
E. Distance between Agent and Predator

```
with open('output.csv', 'w') as output:
    writer = csv.writer(output)
    count=0
    writer.writerow(["Agent_Pos", "Prey_Pos","Pred_Pos", "Prey_Dist","Pred_Dist","Utility"])
    for i in X:
        writer.writerow([i[1],i[2],i[3],i[4],i[5],utility[i[1],i[2],i[3]]])
```

In this, X contains all the data points generated and is inserted into the csv using the above code to generate the dataset. An overview of the generated dataset is :

	A	B	C	D	E	F
1	Agent_Pos	Prey_Pos	Pred_Pos	Prey_Dist	Pred_Dist	Utility
2	1	4	8	2	5	3.65329772
3	1	4	9	2	4	3.60477069
4	1	4	10	2	5	3.65329772
5	1	4	11	2	6	3.44336299
6	1	4	12	2	7	3.62515314
7	1	4	13	2	8	3.61271524
8	1	4	14	2	8	3.40861575

What kind of model are you taking V to be? How do you train it?

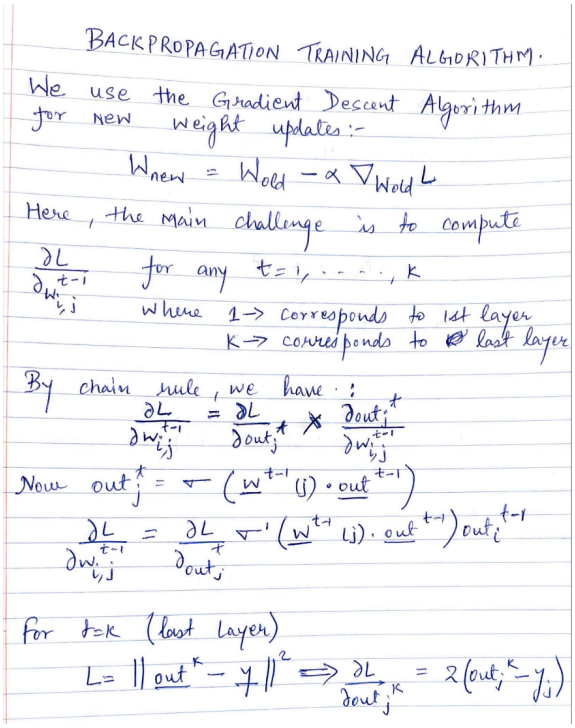
For the model, we have used a neural network. The neural network comprises of 5 hidden layers, one input layer and one output layer. The input layer has 5 nodes which take values from the features. The hidden layers have 6 nodes each. The output layer is just a single node.

We experimented with different kinds of activation functions for our hidden layers 1,2,3 and 4. We tried to maintain a linear activation function for our last layer as we saw this as a regression problem and ultimately, we have to predict a value for utility. For internal layers we experimented with tanh(x), sigmoid(x), linear(x) and RelU(x). After experimenting we decided on using the tanh(x) as activation function for our Neural Network model. This function is only used as activation function for the inner layers, for the last layer we have used the linear activation function to get a regression predicted value.

Training the Network: The basis of training the neural network is the gradient descent algorithm. If ‘W’ represents all collection of weights in the model, then we can find better/ more optimum weights which minimize our loss using the equation –

$$W_{\text{new}} = W_{\text{old}} - \alpha \nabla_{W_{old}} L,$$

Here, $\nabla_{W_{old}} L$ is the collection of derivatives of the loss with respect to each of the weights and alpha (α) is some constant step size. Here choice of alpha determines the rate of descent toward the minima of the loss function. If we take alpha too big there is a high chance to miss the minima of the loss function. So, alpha must be taken relatively small.



BACKPROPAGATION TRAINING ALGORITHM.

We use the Gradient Descent Algorithm for new weight updates :-

$$W_{new} = W_{old} - \alpha \nabla_{W_{old}} L$$

Here, the main challenge is to compute

$\frac{\partial L}{\partial w_{i,j}^{t-1}}$ for any $t = 1, \dots, K$

where 1 → corresponds to 1st layer
K → corresponds to last layer

By chain rule, we have :

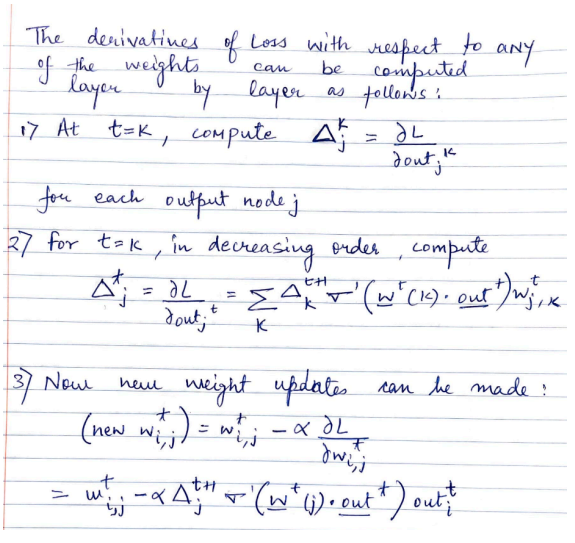
$$\frac{\partial L}{\partial w_{i,j}^{t-1}} = \frac{\partial L}{\partial out_j^t} \times \frac{\partial out_j^t}{\partial w_{i,j}^{t-1}}$$

Now $out_j^t = \sigma\left(\underline{w}^{t-1}(j) \cdot \underline{out}^{t-1}\right)$

$$\frac{\partial L}{\partial w_{i,j}^{t-1}} = \frac{\partial L}{\partial out_j^t} \sigma'\left(\underline{w}^{t-1}(j) \cdot \underline{out}^{t-1}\right) out_i^{t-1}$$

For $t = K$ (last layer)

$$L = \| \underline{out}^K - \underline{y} \|^2 \Rightarrow \frac{\partial L}{\partial out_j^K} = 2\left(out_j^K - y_j\right)$$



The derivatives of Loss with respect to any of the weights can be computed layer by layer as follows :

1) At $t = K$, compute $\Delta_j^K = \frac{\partial L}{\partial out_j^K}$

for each output node j

2) for $t = K$, in decreasing order, compute

$$\Delta_j^t = \frac{\partial L}{\partial out_j^t} = \sum_k \Delta_k^{t+1} \sigma'\left(\underline{w}^t(k) \cdot \underline{out}^t\right) w_{j,k}^t$$

3) Now new weight updates can be made :

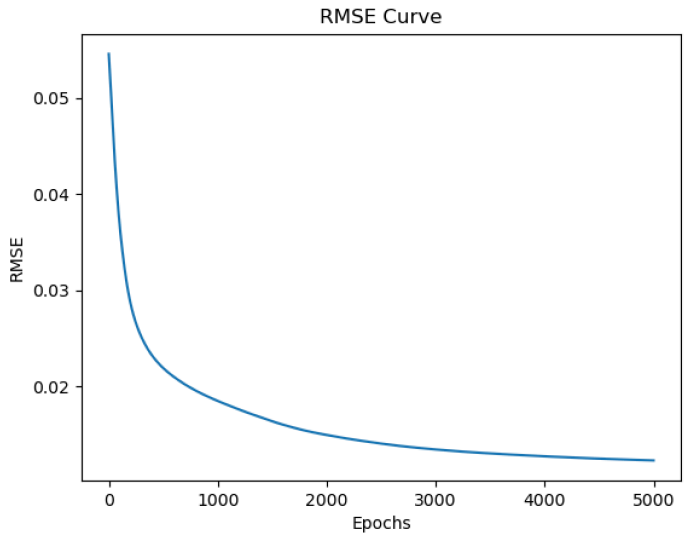
$$(new\ w_{i,j}^t) = w_{i,j}^t - \alpha \frac{\partial L}{\partial w_{i,j}^t}$$

$$= w_{i,j}^t - \alpha \Delta_j^{t+1} \sigma'\left(\underline{w}^t(j) \cdot \underline{out}^t\right) out_i^t$$

Is overfitting an issue here?

We believe that overfitting is not an issue here. We will always be querying for a set of states which will belong to the set of 125000 states present in the dataset. Moreover, our observation here was that this dataset is exhaustive, and we will necessarily not be querying anything outside this dataset. So, overfitting won't be an issue here.

How accurate is V?



The accuracy of the V model is high as the model is trained till RMSE between predicted y value and training value is very low. As we can infer from the graph that RMSE is around 0.02. That means that the expected error between the predicted value and true value will be varying by around 0.02. So, for example if the true value of Utility of a query is 8, then the model can predict values in the range of 7.98 to 8.02 for this data point. Hence, we conclude that the accuracy of V is high.

How does its performance stack against the U∗ agent?

```
Steps for U* = 9.6 Survivability = 100%
Steps for V* = 10.8 Survivability = 100%
(base) viswajithmenon@Viswajiths-MacBook-Air ~ %
```

Simulate an agent based on U_partial, in the partial prey info environment case from Project 2, using the values of U* from above. How does it compare to Agent 3 and 4 from Project 2? Do you think this partial information agent is optimal

Utility Implementation for partial info setting–

In this setting the agent is aware only of the position of the predator. This change forces us to recalculate the utilities from our original 'utility' dictionary as the state for which we were checking utility for included the position of the prey.

Formula used to determine the utility of a neighbor of the agent –

$$U_{\text{partial}}(s_{\text{agent}}, s_{\text{predator}}, \underline{p}) = \sum_{s_{\text{prey}}} p_{s_{\text{prey}}} U^*(s_{\text{agent}}, s_{\text{predator}}, s_{\text{prey}}).$$

In the equation given above the 'p' denotes a belief vector of 50 values. Each value in this vector represents the probability of the prey being in node 'i' of the graph, where 'i' is the index position in the vector. (i ranges from 1-50)

Probability / Belief Calculations:
Here when the agent spawns at a node in the graph, 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).

We then survey according to the rules of agent 3.

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

(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.

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

$= P(\text{Prey in 1 NOW}) \cdot P(\text{Prey in 1 next} / \text{prey in 1 now})$
$+ P(\text{prey in 2 Now}) \cdot P(\text{Prey in 1 next} / \text{prey in 2 now})$
$+ P(\text{Prey in 3 Now}) \cdot P(\text{Prey in 1 next} / \text{Prey in 3 Now})$
$+ \dots$
$+ P(\text{Prey in 20 Now}) \cdot P(\text{Prey in 1 next} / \text{Prey in 20 now})$
$+ \dots$
$+ P(\text{Prey in 49 Now}) \cdot P(\text{Prey in 1 next} / \text{Prey in 49 now})$
$+ P(\text{Prey in 50 NOW}) \cdot P(\text{Prey in 1 next} / \text{Prey in 50 Now})$

So, According to this, the transition probabilities for the Agent after they prey moves are:

$P(\text{prey in 'K' Node next}) =$

$\sum_{i=01}^{50} P(\text{Prey in 'i' NOW}) \cdot P(\text{Prey in 'K' next} / \text{prey in 'i' now})$

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.

```
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
```

U* Agent Implementation for partial information setting–

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 Upartial_Agent(adjacency_list):
    count_prey_pos = 0
    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)
```

Once we have populated the vector of beliefs ‘p’ we run the equation to calculate the utilities of the neighbors of the agent.

Based on the survey result we then update the ‘probability_table’ the same way we had done for agent 3 in project 2.

The utilities of the neighbors of the agent are then calculated using this updated ‘probability_table’.

```
upartial_dict = {}
upartial_dict.clear()
for nbr in agent_nbrs:
    summatn = 0
    l=[]
    for probab_index in range(len(probability_table)):
        summatn += probability_table[probab_index] * utility[nbr,probab_index + 1,Predator1.position]
    upartial_dict[nbr,Predator1.position] = summatn
    #VISH
    l.append(nbr)
    l.append(Predator1.position)
    for a in range(len(probability_table)):
        l.append(probability_table[a])
    l.append(upartial_dict[nbr,Predator1.position])
    Partial_Dataset.append(l)
    #VISH
#Upartial values calculated -> Now move the agent to the lowest Upartial position
minUtil = 90000
for key, val in upartial_dict.items():
    if(upartial_dict[key] < minUtil):
        [ag,pr] = key
        mn_nbr = ag

Agent1.position = mn_nbr
```

As shown in the screenshot above we evaluate the utilities of the neighbors by summing over all the prey positions and finding the product of the probability of a particular node having the prey with the previously calculated utility of that state.

In every timestep when the prey moves, we call the probability_distribution2 function to update the transition probabilities after each iteration.

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)
```

This process is repeated until either the agent wins or the predator wins.

Simulate an agent based on Upartial, in the partial prey info environment case from Project 2, using the values of U^* from above. How does it compare to Agent 3 and 4 from Project 2? Do you think this partial information agent is optimal?

The number of steps given below is the average number of steps taken by the agent to catch the prey in 3000 runs on one graph.

```
Steps for U_Partial = 28.43 Survivability = 98.73%
Steps for Agent 3 project 2 = 31.87, Survivability = 78.34%
Steps for Agent 4 project 2 = 38.51, Survivability = 86.521%
(base) viswajithmenon@Viswajiths-MacBook-Air ~ %
```

U_Partial Agent –

- Survivability – 98.73%
- Number of steps taken – 28.43

Agent 3 from project 2 –

- Survivability – 78.34%
- Number of steps taken – 31.87

Agent 4 from project 2 –

- Survivability – 86.521%
- Number of steps taken – 38.51

We think that the Upartial Agent is optimal as it takes the lowest number of steps to capture the prey while keeping its survivability high.

Model V Partial:

How do you represent the states agent, predator, p as input for your model? What kind of features might be relevant?

Dataset for V Partial Model:

The data for this model is generated while running UPartial Agent. When we execute UPartial agent we keep appending the Agent Position, Predator Position and the Vector of Probabilities in each column as a feature. We also append the calculated utility during the execution of Upartial Agent to the dataset. So, here we are using the Agent Position, Predator Position and the value of each of the belief as features for the VPartial Model.

```
upartial_dict[nbr,Predator1.position] = summatn
#VISH
l.append(nbr)
l.append(Predator1.position)
for a in range(len(probability_table)):
    l.append(probability_table[a])
l.append(upartial_dict[nbr,Predator1.position])
Partial_Dataset.append(l)
```

What kind of model are you taking Vpartial to be? How do you train it?

We have implemented the Vpartial Model as a neural network. We have used the same structure as implemented in Model V. The changes here are that we have to deal with a total of 52 features and the number of hidden nodes per layer is 8. Here, since there more features are present in the dataset, we have increased the number of nodes per layer to capture the relationship between features and their complexity. The activation functions used are similar. The internal layers use Tanh(x) as their activation function and the last layer uses a linear activation function. This is done to predict real values of utility as we see this as a regression problem. As training the V Model before we use the forward pass and backwards pass to calculate optimal weights for each feature and a reasonable bias. We update

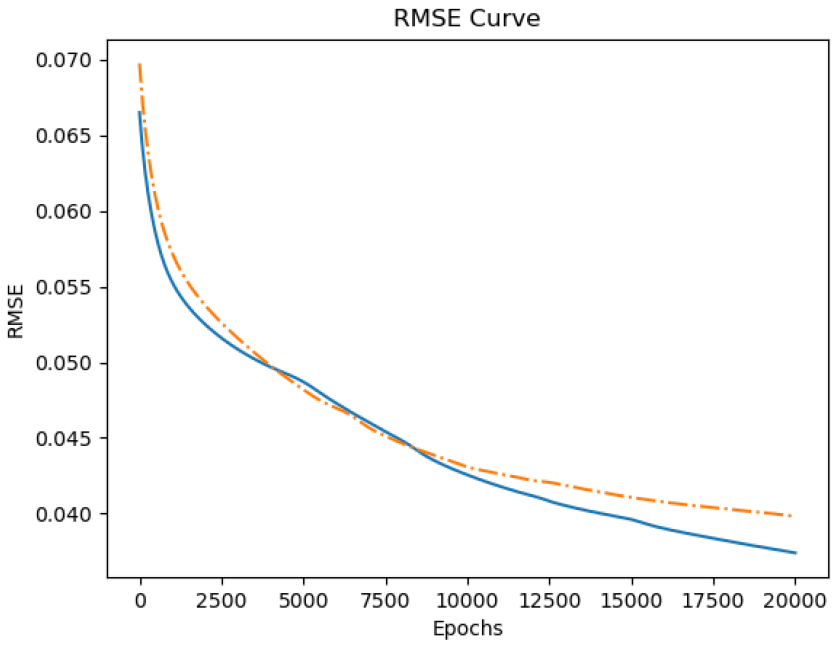
these as per the gradient descent algorithm. After successful runs and minimizing the RMSE, we generate a dictionary of layer-wise weights to get the final optimized weights. After this, for every prediction for Model Vpartial we call this dictionary and do a forward pass with the data to predict the value of expected utility for a particular datapoint.

Is overfitting an issue here? What can you do about it?

Overfitting can become an issue here. This is because the queries here can be seen as partial queries and our dataset here is not exhaustive. This means that we don't have all the data (X-Values) for which the queries will be made from the model while running Vpartial agent. We can prepare for this scenario by making our model optimal by using a train-test split of 80% and 20%. In this way our model will not be in an overfit state.

```
dataset = shuffle(pd.read_csv("output_partial.csv"))

dataset.columns = ["Agent_Pos", "Pred_Pos","Belief1", "Belief2","Belief3","Belief4","Belief5",
"Belief6","Belief7","Belief8","Belief9","Belief10","Belief11","Belief12","Belief13","Belief14",
"Belief15","Belief16","Belief17","Belief18","Belief19","Belief20","Belief21","Belief22","Belief23",
"Belief24","Belief25","Belief26","Belief27","Belief28","Belief29","Belief30","Belief31","Belief32",
"Belief33","Belief34","Belief35","Belief36","Belief37","Belief38","Belief39","Belief40","Belief41",
"Belief42","Belief43","Belief44","Belief45","Belief46","Belief47","Belief48","Belief49","Belief50","Utility"]



x = dataset[["Agent_Pos", "Pred_Pos","Belief1", "Belief2","Belief3","Belief4","Belief5",
"Belief6","Belief7","Belief8","Belief9","Belief10","Belief11","Belief12","Belief13","Belief14",
"Belief15","Belief16","Belief17","Belief18","Belief19","Belief20","Belief21","Belief22","Belief23",
"Belief24","Belief25","Belief26","Belief27","Belief28","Belief29","Belief30","Belief31","Belief32",
"Belief33","Belief34","Belief35","Belief36","Belief37","Belief38","Belief39","Belief40","Belief41",
"Belief42","Belief43","Belief44","Belief45","Belief46","Belief47","Belief48","Belief49","Belief50"]]
y = dataset[["Utility"]]

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size= 0.2)

x_train = pre.StandardScaler().fit_transform(x_train)
x_test = pre.StandardScaler().fit_transform(x_test)

#x_train = x.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()
```

To avoid overfitting, while training the model we keep track of the error between the test-split data and the model's predicted value of data. We notice a steady decrease similar to the root mean squared error between model's predicted value and training data. We keep training till we see a decrease and similar trend in both curves. At one point, the error between the model's predicted value and test-split data begins to increase. This is the region where we must stop training the model to avoid overfitting.



In the above graph the blue line represents the RMSE between training data and predicted value of model and the dotted orange line represents the RMSE between test data and predicted value of model. We noticed that after around 10,000 epochs the trend between the two curves changes, so that is the region where we have to stop training the model to avoid overfitting.

How accurate is Vpartial? How can you judge this?

Vpartial is reasonably accurate. We can infer this from the RMSE curve. When we terminate training of model, the RMSE is very low. A low value of RMSE means that the deviation of the predicted value of utility from true value of utility is also low. For instance, in the above graph, the RMSE is around 0.045, so if we are trying to predict the utility whose true value is 10 then the model can predict values in the range of 9.955 to 10.045. Using this we conclude that the accuracy of the model is acceptable.

Is Vpartial more or less accurate than simply substituting V into equation (1)?

Yes, Vpartial is approximately as accurate as simply substituting V into equation1. We have tested this in the function: compare_Vpartial(adjacency_list,agentPos,predPos,probability_table). In this function we pass in the adjacency list, agent position, predator position and the table of beliefs about the prey position. Then we calculate the difference between

Vpartial value and the value generated by simply substituting V and verify that their difference is low which signifies that they are approximately same.

```
def compare_Vpartial(adjacency_list,agentPos,predPos,probability_table):
    value_of_Vpartial = get_util_from_V_Partial(agentPos, predPos, probability_table)

    summatn = 0
    predDis = djikstra(adjacency_list, agentPos, predPos)
    for probab_index in range(len(probability_table)):
        preyDis = djikstra(adjacency_list, agentPos, probab_index + 1)
        summatn += probability_table[probab_index] * get_util_from_V(agentPos, probab_index + 1, predPos, preyDis, predDis)
    value_from_simply_plugging_V = summatn
    diff = abs(value_from_simply_plugging_V - value_of_Vpartial)
    print("Difference : ", diff)
```

Simulate an agent based on Vpartial. How does it stack against the Upartial agent?

Agent Based on Vpartial Model:

The Vpartial agent spawns at a random location in the graph and does surveying for prey position similar to agent 3 in project 2. After each survey it updates its belief/probability table. This table is also update every time the agent moves with the transition probabilities using probability_distribution2 function as in Upartial agent and agent 3 project 2.

```
def VPartial_Agent(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)

    #Clearing probability table
    probability_table.clear()
    #Inserting Raw probabilities
    for x in range(no_of_nodes):
        probability_table.append(1/no_of_nodes)

    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))

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

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

```
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 the agent whenever it has to move performs a query from the function *get_util_from_V_Partial(agentPos,predPos,probability_table).* This function takes arguments as the agent position, predator position and the belief/probability array of the agent about where the prey is as the input. Inside this function, we convert the arguments to match the input features for our model so that we can perform a forward pass with the optimal weights from the dictionary of weights generated by the Vpartial model.

```
def get_util_from_V_Partial(agentPos,predPos,probability_table):

    print("Predicted Values")
    var = pd.Series([agentPos,predPos,probability_table[0],probability_table[1],probability_table[2],probability_table[3],
    probability_table[4],probability_table[5],probability_table[6],probability_table[7],probability_table[8],probability_table[9],
    probability_table[10],probability_table[11],probability_table[12],probability_table[13],probability_table[14],probability_table[15],
    probability_table[16],probability_table[17],probability_table[18],probability_table[19],probability_table[20],
    probability_table[21],probability_table[22],probability_table[23],probability_table[24],probability_table[25],
    probability_table[26],probability_table[27],probability_table[28],probability_table[29],probability_table[30],
    probability_table[31],probability_table[32],probability_table[33],probability_table[34],probability_table[35],
    probability_table[36],probability_table[37],probability_table[38],probability_table[39],probability_table[40],
    probability_table[41],probability_table[42],probability_table[43],probability_table[44],probability_table[45],
    probability_table[46],probability_table[47],probability_table[48],probability_table[49]],
    index = ["Agent_Pos", "Pred_Pos","Belief1", "Belief2","Belief3","Belief4","Belief5",
    "Belief6","Belief7","Belief8","Belief9","Belief10","Belief11","Belief12","Belief13","Belief14",
    "Belief15","Belief16","Belief17","Belief18","Belief19","Belief20","Belief21","Belief22","Belief23",
    "Belief24","Belief25","Belief26","Belief27","Belief28","Belief29","Belief30","Belief31","Belief32",
    "Belief33","Belief34","Belief35","Belief36","Belief37","Belief38","Belief39","Belief40","Belief41",
    "Belief42","Belief43","Belief44","Belief45","Belief46","Belief47","Belief48","Belief49","Belief50"])
    # print("va2 : ", va2)

    #Accessing weights from model
    output_weight = v_partial_weight_dict["output_weight"]
    weight_la1 = v_partial_weight_dict["weight_la1"]
    weight_la2 = v_partial_weight_dict["weight_la2"]
    weight_la3 = v_partial_weight_dict["weight_la3"]
    weight_la4 = v_partial_weight_dict["weight_la4"]
    weight_la5 = v_partial_weight_dict["weight_la5"]
```

```
# Hyperbolic Tangent Activation function
def hyperbolic_tanh(x):
    return (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))

# Hyperbolic derivative
def derivative_hyperbolic(x):
    return 1 - hyperbolic_tanh(x) * hyperbolic_tanh(x)

# Linear Activation Function
def linear_activation(x):
    return x

# Linear derivative
def derivative_linear(x):
    return 1


layer_1 = np.dot(var, weight_la1)
layer_1_out = hyperbolic_tanh(layer_1)
#print("layer_1 : ", layer_1)

layer_2 = np.dot(layer_1_out, weight_la2)
layer_2_out = hyperbolic_tanh(layer_2)

layer_3 = np.dot(layer_2_out, weight_la3)
layer_3_out = hyperbolic_tanh(layer_3)

layer_4 = np.dot(layer_3_out, weight_la4)
layer_4_out = hyperbolic_tanh(layer_4)

layer_5 = np.dot(layer_4_out, weight_la5)
layer_5_out = linear_activation(layer_5)

output = np.dot(layer_5_out, output_weight)
final_out = linear_activation(output)

print("X-Val : ", var)
#print("Actual Val : 5.204084483")
print("Predicted Value : ", final_out)
return final_out
```

We query for every action/neighbor the agent moves to get the predicted utility value and then move to the neighbor with the lowest value of utility.

```
agent_nbrs = []
for x in adjacency_list[Agent1.position]:
    agent_nbrs.append(x)

min_Util = 90000000
for nbr in agent_nbrs:
    util_from_vPartial = get_util_from_V_Partial(nbr, Predator1.position, probability_table)
    if(util_from_vPartial < min_Util):
        Agent1.position = nbr
```

The number of steps given below is the average number of steps taken by the agent to catch the prey in 3000 runs on one graph.

U_Partial Agent –

- Survivability – 98.73%
- Number of steps taken – 29.65

V_Partial Agent –

- Survivability – 98.68%
- Number of steps taken – 36.85