CS 520 - Project 3: Better, Smarter, Faster

Submitted by Aditya Manelkar (am3369) and Anmol Arora (aa2640)

Design and Algorithms

The Circle of Life

Description

The graph creation, prey movement, distracted predator movement, and functions that determined movements for agents 1 through 4 were re-used from Project 2. A brief overview is covered in the codes section.

Important Codes

graph_utils.py: It has the logic for graph creation and all necessary methods to extrapolate data (like shortest path lengths, etc.) from said graph.
prey.py: Contains the prey spawn logic and movement logic for the prey.
predator.py: Contains the prey spawn logic and movement logic for the easily distracted predator.
agent.py: Contains agent spawn and movement logic, and holds key data required in a partial environment setting like the prey position belief vector and transition matrix.
probability.py: Used in agent.py, it has integral parts of code for updating beliefs in the partial environment setting.
data.py: Used for plotting comparison charts and graphs (tweaked now for comparing the new agents)

Better, Smarter, Faster

Description

For the current project, the focus is on creating and comparing different models and comparing them against agents from project 2 or against each other. The individual agents running based on different models will be touched on in detail, but an overview of the codes over and above those from project 2 is given below.

Important Codes

final.ipynb: The notebook that has all the models (for new agents) we have worked on for this project.

visualize.ipynb: Used to visualize the movement of the prey, predator and agent at each timestep to juxtapose the U* agent against agents 1 and 2 from project 2. (While we have used networkx here, it is strictly done in the capacity of only displaying/visualizing graphs. No important functions like shortest path distances or even graph structures were leveraged here.)
pickle: The library was used to create some sort of a database of certain known values like optimal utilities, policies, etc., and the data from PKL files were retrieved into dictionaries to avoid re-computation each time the kernel was restarted.

Test Cases

Description

We stuck by the following guidelines:

1) For comparing agents, we made sure to divide a test batch into 30 runs with 100 trials per run - which amounted to a total of 3000 games per agent.
2) Unlike project 2 though, we have the same graph used but we use different starting states for each of the runs.
3) We ran each game for a maximum of 5000 time steps (as suggested by Dr. Cowan).
4) While comparing the agents (i.e. the new models) against previous agents or even the newer ones, we tried to compare the average success rates and the average number of timesteps each agent took before the game was finished. We did not consider the timeout rates because our game never timed out with 5000 time steps.

Agent 11 - U*

Description

The positions of the agent, prey, and predator in the graph are used to create a Markov Decision Process, in which each state is represented by these positions as we are in the complete information setting and hence we are aware about these parameters at every timestep during the simulation of the game. The agent must be in a position where its position and that of the prey are identical in order to capture the prey, but it must also guard against being captured by the predator (i.e a state where the position of the agent and the position of predator is same.). In this report, we refer to these states as the "good terminal state" and the "bad terminal state," respectively. The agent's objective is to arrive at the desirable terminal state starting from an arbitrary initial state.
The optimal agent uses a lookup against the database of optimal utilities to return a utility. This in turn is used to decide the policy/action to be taken when in a given state (s).

Q. How many distinct states (configurations of predator, agent, prey) are possible in this environment?
The state of the agent is represented by the position of the agent, the position of the prey and the position of the predator in the graph. We know that each of these can have a maximum of 50 values (i.e the agent position can be at any node in the graph and since the total nodes in the graph is 50 => It can take any value between 0 to 49).

The total number of combinations we can generate from this is 50x50x50 = **125,000** (since each agent,prey and predator can have any value between 0 to 50).
Hence, the total distinct states that we can have is **125,000.**

Q. What states s are easy to determine U* for?

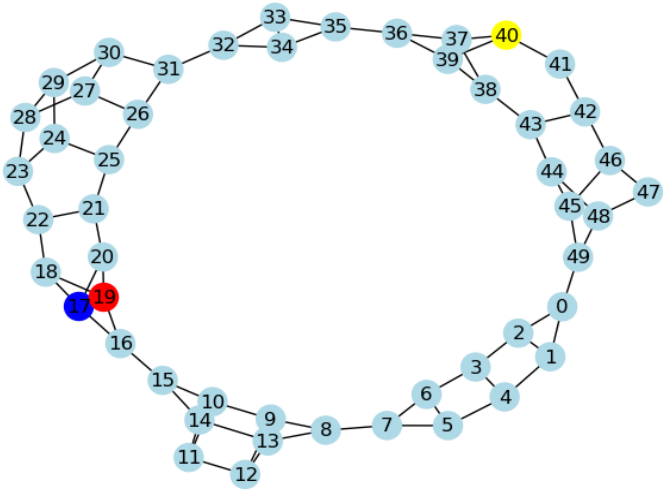
We can determine the utility for terminal states easily.
Case 1: Agent catches the Prey (i.e “Good” Terminal State)
We can set the utility for this state as 0 since the agent has already caught the prey and the minimum expected number of rounds to catch the prey is 0.
Case 2: Predator caught the Agent (i.e “Bad” Terminal State)
We can set the utility for this state as 5000 (random high number other than infinity) since the predator has already caught the agent and the minimum expected number of rounds to catch the prey can’t increase more than 5000 according to the constraint mentioned in Project 2 where the max timesteps the game can run for is 5000 .
Case 3: Agent is right next to the Prey (i.e the distance between agent and prey is equal to 1)
When the agent is right next to the prey the we know that the minimum number of expected rounds to catch the prey must be equal to 1.

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

At every state an agent can take various actions/policies which helps it to transition to the neighbouring state(i.e the state with positions of the agent,prey and predator at the next timestep). The agent must move to the neighboring state which helps it to reach the “good terminal state” in a minimum number of timesteps. We assign utilities to each state, where utility represents the minimum expected number of rounds to catch the prey, in order to determine the optimum course of action. At every state we calculate the utility by comparing the utilities of neighboring states the agent can transition to depending on the actions it has available at that state. The probability of transitioning to a neighboring state is calculated using the transitioning probability of the prey and predator moving to the new position in the neighboring state. The action that gives us the minimum utility is state policy for that state.
We calculate the optimal state utility for every state using the **value iteration** method to solve the bellman equations (i.e non-linear system of equations) for this MDP. This in turn helps us to find optimal state policy for every state.

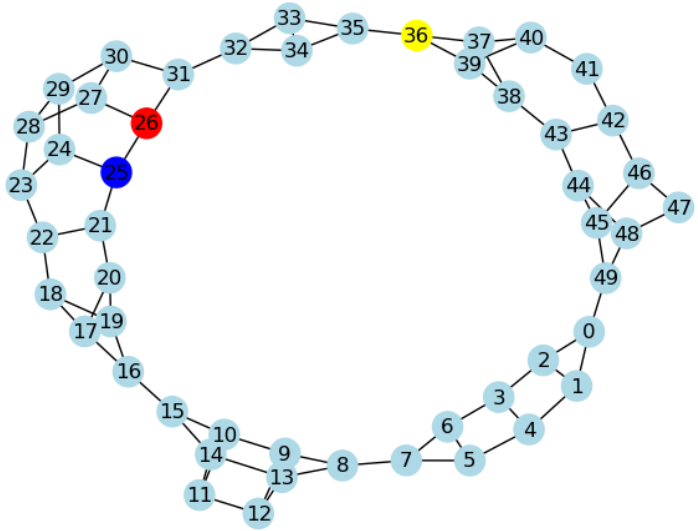
Q: Are there any starting states for which the agent will not be able to capture the prey? What causes this failure?

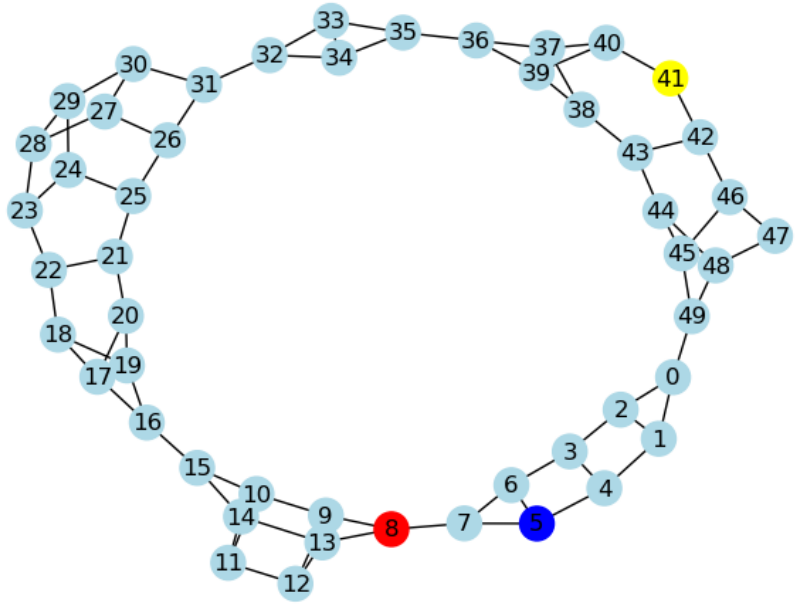
Every neighbor of the agent is also reachable by the predator in one move,in such cases we hope for the predator to be distracted only in those cases we will be able to have a chance to catch the prey. Ex. Run(10)
There is a 60 percent chance that the predator will move to the node where the agent moves. Our only chance of survival is to hope that the predator is distracted.

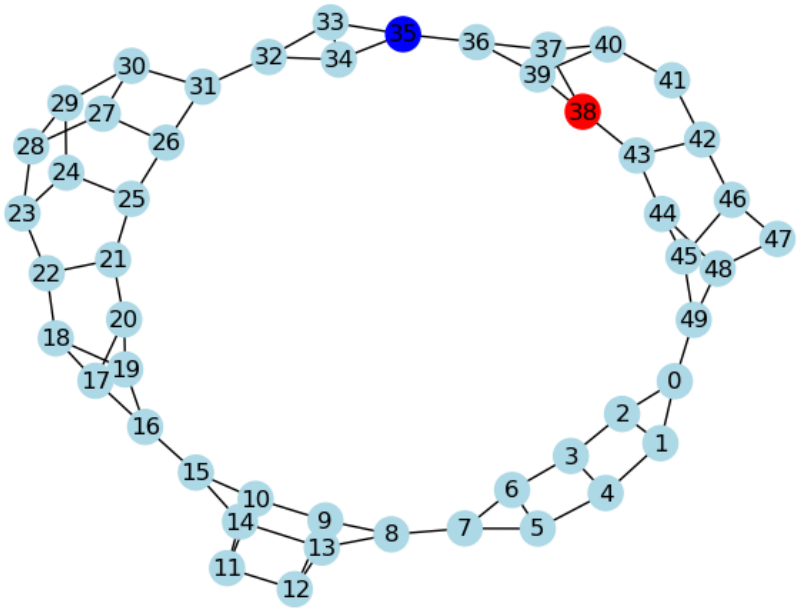


Blue - Agent
Red - Predator
Yellow - Prey

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







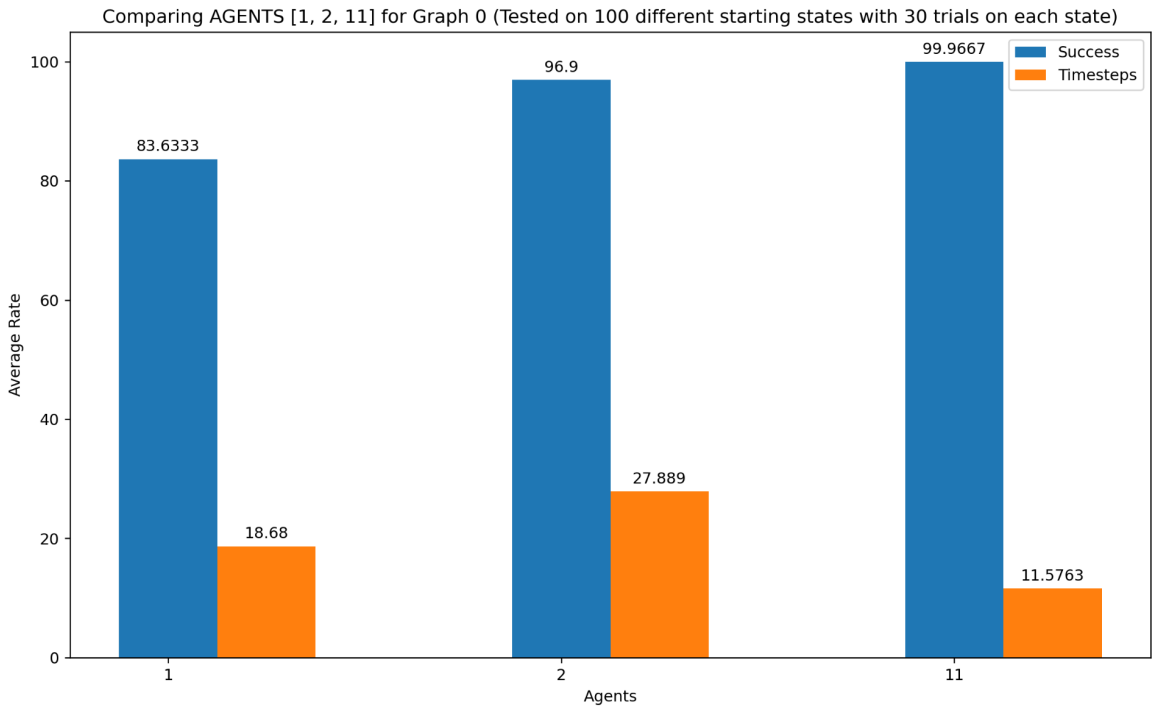
The largest possible value of U* signifies the maximum number of timesteps it took for the agent to catch the prey. Out of all the starting that we have considered, the maximum amount of timesteps the agent took to catch the prey was when the predator was right behind the agent and in order to catch the prey the agent had to traverse almost the whole graph.
Timestep for this particular state turns out to be 24 as compared to the average of around 11 timesteps.

Important Code Snippets

Final.ipynb: value_iteration(): The method that calculates the optimal utility for every state.
agent.Agent.move_utility(): The method that performs the entire priority based movement.

Q: 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?

We generally see that our Agent 11 outperforms Agents 1 and 2 from project 2 in the complete information setting as can be seen below.



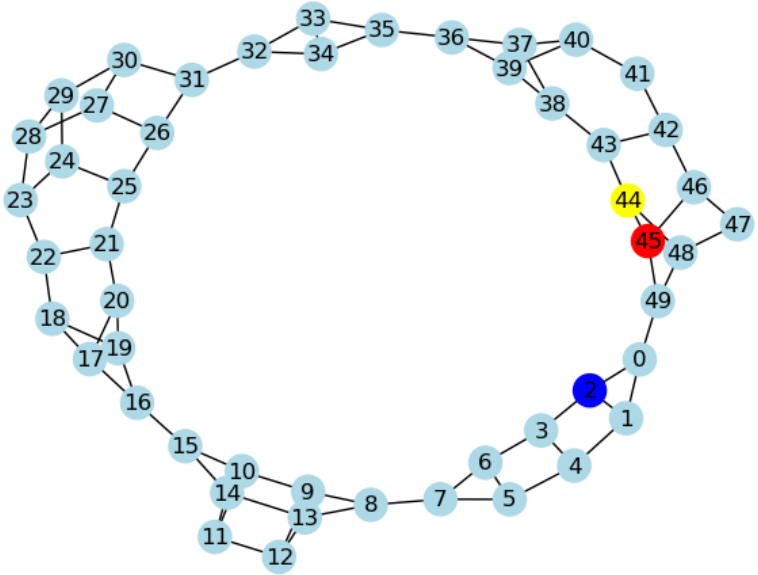
Agent	Mean Success Rate	Mean Timesteps
Agent 1 - Complete Info	83.633	18.68
Agent 2 - Complete Info	96.9	27.889
Agent 11 - U*	99.9667	11.5763

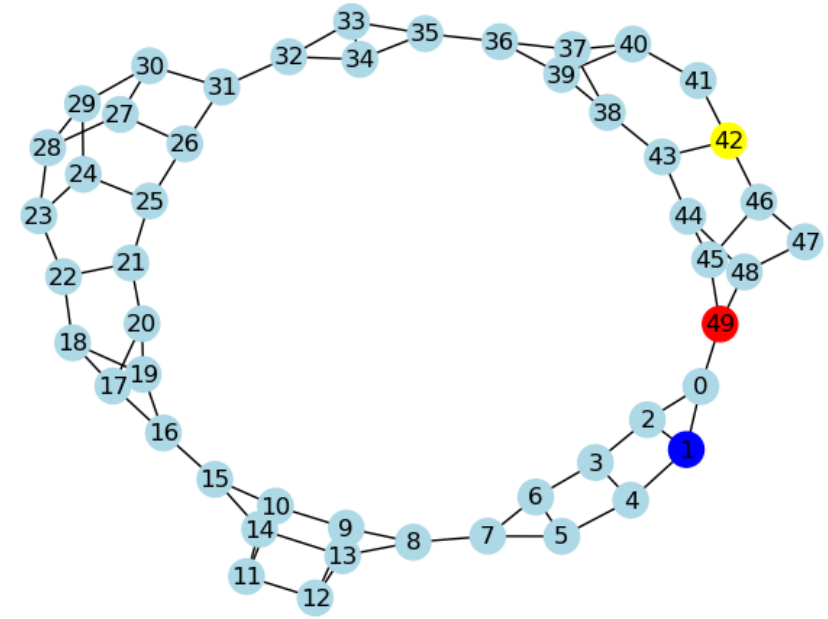
Q. Are there states where the U* agent and Agent 1 make different choices? The U* agent and Agent 2? Visualize such a state, if one exists, and explain why the U* agent makes its choice.

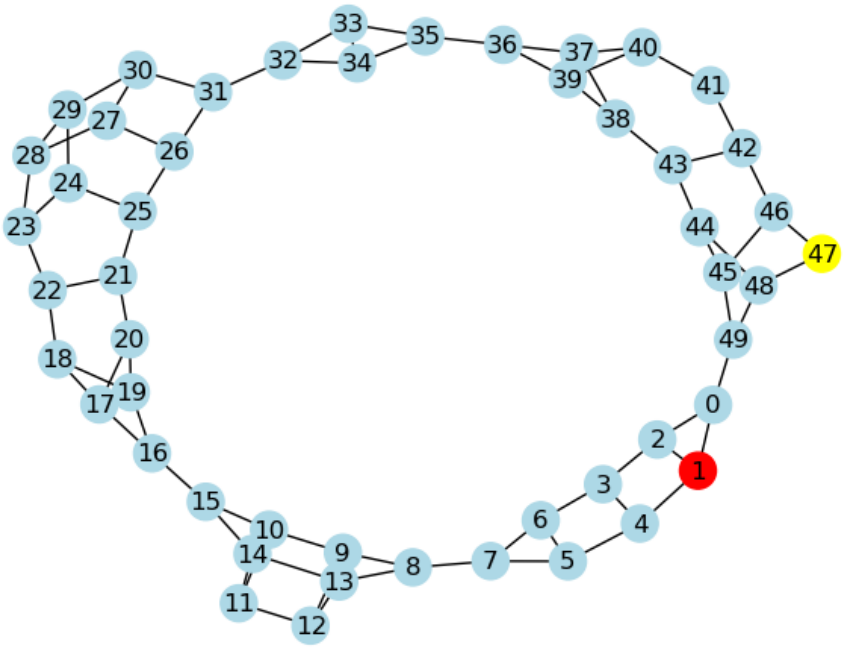
In cases where Predator is between the shortest path from agent to prey -> Our agents 1 and 2 fail to detect that it is dangerous to go in the direction of the shortest path to prey and hence they think moving close to the prey is a good decision but in reality they get in trouble when the predator is to close for comfort and end up getting caught.
Agent U* (11) looks at the bigger picture here and identifies that going in the direction of the prey is not beneficial and hence takes the longer route to catch the prey which at the end turns out to be the right approach as it is able to catch the prey.
The reason why Agent U* looks at the bigger picture is because out of all the neighbors the neighbor where the distance between the Agent and Predator is decreasing from the current distance between them(although the the distance between the Agent and the Prey might be decreasing for that state) have high utility value as compared to the one where the distance between them increases (although the distance between the Agent and the Prey might be increasing for that state) and since the agent always moves to the state with minimum utility it always chooses the path to catch the prey that takes it away from the predator.

In the report we have attached images of the initial state, state at the middle of the game and the state at the end of the game.
The whole visualization is done in **visualise.ipynb**.

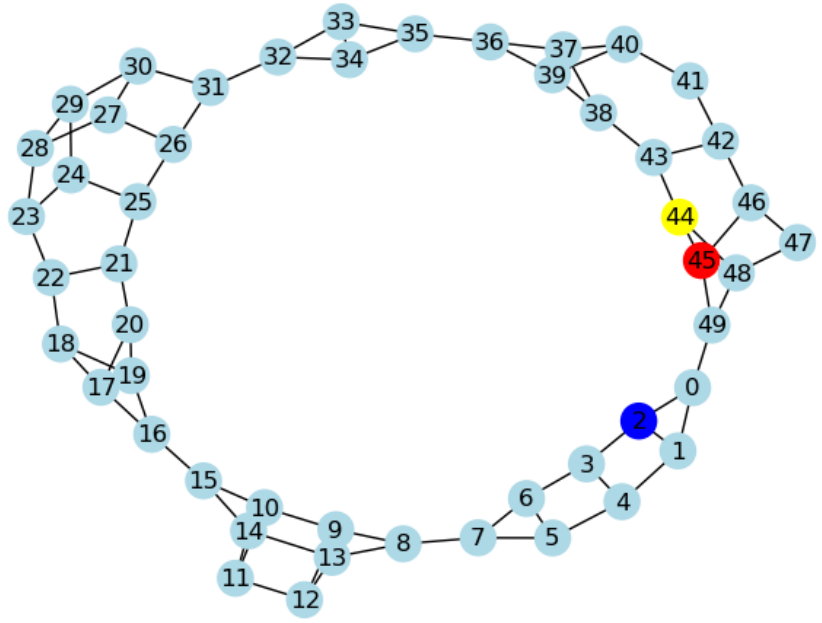
Agent 1

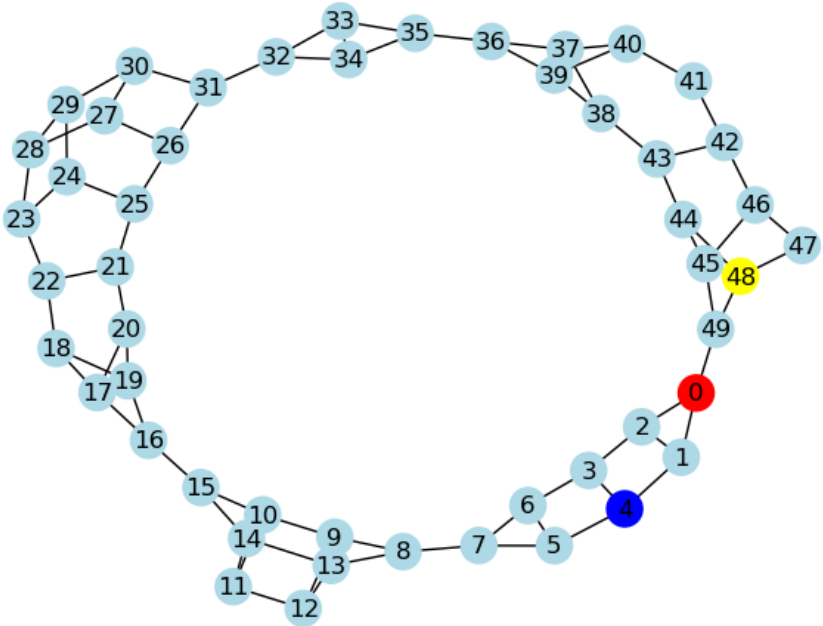


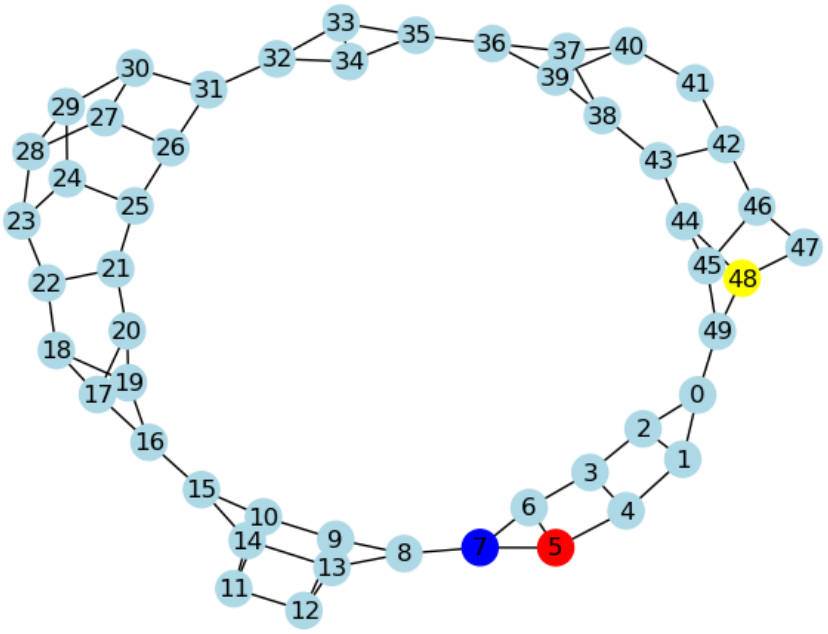

33
35
36
37
40
32
34
39
41
30
31
38
29
27
42
26
43
28
24
46
25
44
23
47
45
21
48
22
49
20
18
0
19
17
2
16
1
3
15
6
4
10
14
9
8
7
5
13
11
12

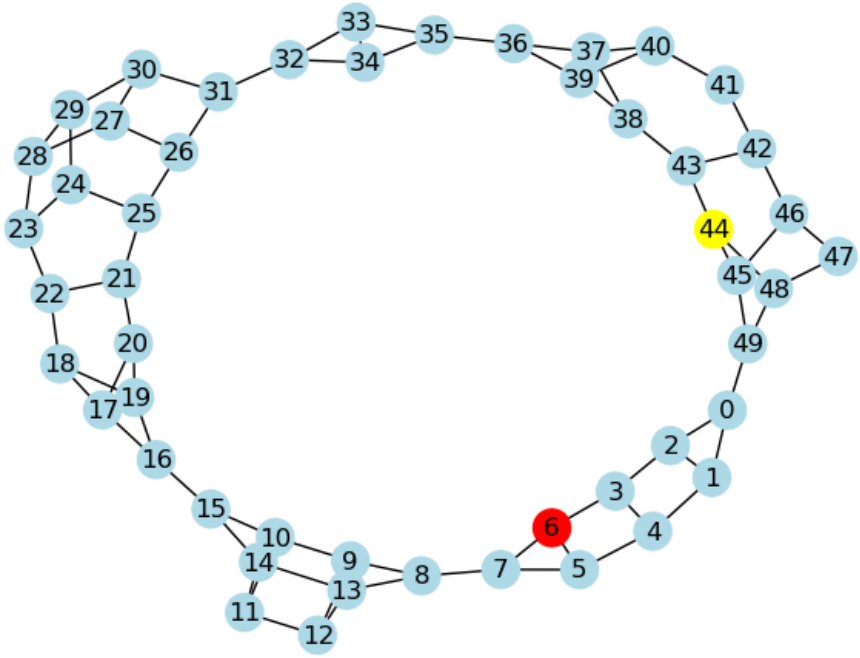

33
35
36
37
40
32
34
39
41
30
31
38
29
27
42
26
43
28
24
46
25
44
23
47
45
21
48
22
49
20
18
0
19
17
2
16
1
3
15
6
4
10
14
9
8
7
5
13
11
12

Agent 2

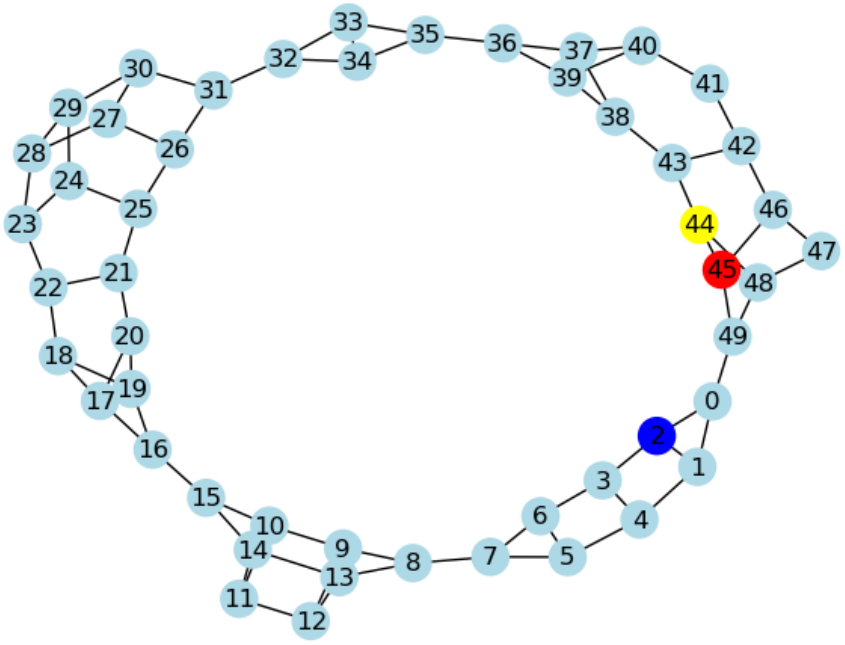

33
35
36
37
40
32
34
39
41
30
31
38
29
27
42
28
26
43
24
46
25
44
23
47
45
21
48
22
49
20
18
0
19
17
2
16
1
3
15
6
10
4
14
9
8
7
5
13
11
12

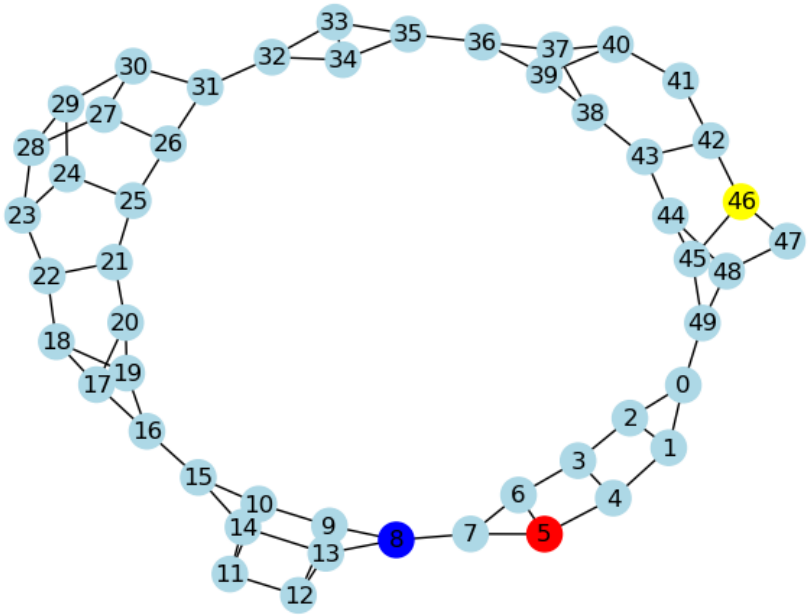

33
35
36
37
40
32
34
39
41
30
31
38
29
27
42
28
26
43
24
46
25
44
23
47
45
21
48
22
49
20
18
0
19
17
2
16
1
3
15
6
10
4
14
9
8
7
5
13
11
12

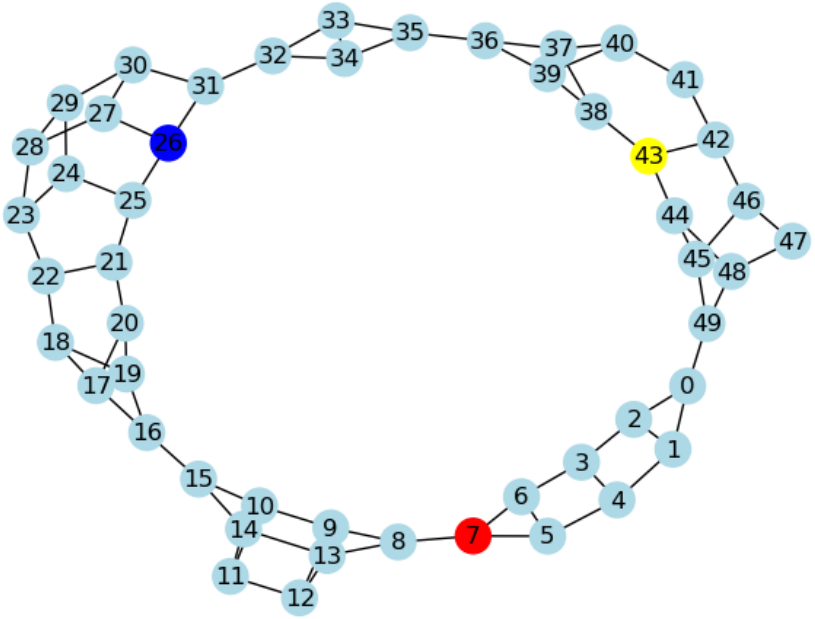

33
35
36
37
40
32
34
39
41
30
31
38
29
27
43
42
28
26
24
46
44
25
23
47
45
48
21
22
49
20
18
0
19
17
2
16
3
1
15
6
4
10
9
14
8
7
5
13
11
12

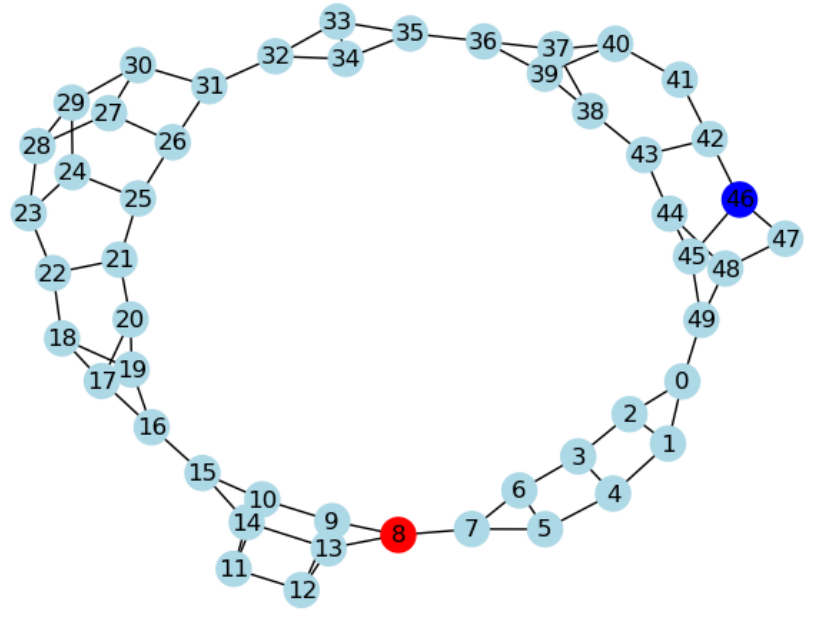

33
35
36
37
40
32
34
39
41
30
31
38
29
27
43
42
28
26
24
46
44
25
23
47
45
48
21
22
49
20
18
0
19
17
2
16
1
3
15
6
4
10
9
14
8
7
5
13
11
12

Agent U* (Agent 11) - Utility based






33
35
36
37
40
32
34
39
41
30
31
38
29
27
42
26
43
28
24
46
44
25
23
47
45
21
48
22
49
20
18
0
19
17
2
16
1
3
15
6
4
10
14
9
8
7
5
13
11
12


33
35
36
37
40
32
34
39
41
30
31
38
29
27
42
26
43
28
24
46
44
25
23
47
45
21
48
22
49
20
18
0
19
17
2
16
1
3
15
6
4
10
14
9
8
7
5
13
11
12

Agent 13 - Model V

Description

The Model V based agent uses a prediction logic rather than a lookup against the database of optimal utilities to return a utility. This in turn is used to decide the policy/action to be taken when in a given state (s). Just like in the previous U* case, the agent that uses this model does so in the complete information setting.

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

One thing that we did initially for the states, was to 1-hot encode them into vectors of length 50 (the number of possible nodes) and then flatten the input (so a vector of length 150). But what we realized was that the feature that would help us the most in such a model would actually be the vector of distances from the agent to the prey and predator.

So the states were input into the model V as a pair of distances from the agent to the predator and the agent to the prey.

Some benefits we felt this would bring about:

1. We would have fewer “states” to train on. Because there would be at max an order of $O(n^2)$ combinations of inputs for (n) nodes over the possible $O(n^3)$ combinations of inputs in the case of 1-hot encoding.
2. We knew we had to fit the model for a specific graph. So in such a case, it would be better to get more precise utilities based on the distances.
3. And even if anytime we would be required to have to train on multiple graphs, distances could always be good measures over nodes (given that the structure/edges of the graph would change).

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

The model we have taken V to be is a Neural Network.

For the training, we do the following:

1. The NN takes 2 inputs for every training data point (distance to the prey and distance to the predator) and derives an output based on forward propagation through activation functions on the dot products of vectors of inputs and weights and some biases.
2. It then finds the squared error (i.e. Loss) of the prediction and the training output and backpropagates it using stochastic gradient descent to update all the weights and biases at each layer.

Few peculiarities about the training:

1. The training data is standardized to get better convergence to output values.
2. The learning rate (or step size) is decayed for every data point in a batch by some decaying factor. And to avoid biases based on data ordering, the training data points in each batch are randomly ordered.

3. The model consists of 2 nodes in the input layer, followed by 4 nodes in one hidden layer and 2 nodes in another, and finally an output layer with one node.
4. What we did end up using is a tanh activation for the hidden layers and a linear activation on the output layer.

Q: Is overfitting an issue here?

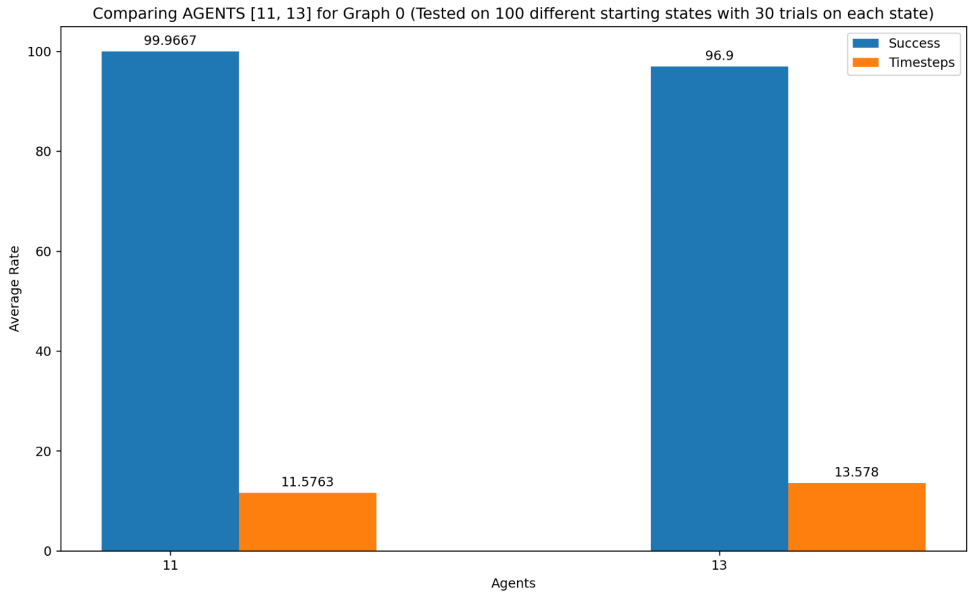
The model would by definition overfit here, but not as much. If the logic to calculate U* is consistent on the basis of distances to the predator and the prey on other graphs as well, then this model would do well. The same would not be applicable if the inputs for the model were just the positions of the agent, predator, and prey, and the model would overfit on a specific graph structure used in training.

Q: How accurate is V?

The model makes predictions pretty accurately, with an average error of 0.120954 (even for a small number of epochs). Also, the success rate of the agent is good without compromising on the average number of timesteps taken by the agent to catch the prey (or for the game to end) compared to the U^*_l agent - indicating the model has not overfit to running from the predator.

Q: How does its performance stack against the U* agent?

In general the model does very well but does *slightly* worse compared to U*. Having said that, it shows how the model already learns pretty well without a concrete formula to end up with success rates and average timesteps very close to U*.



Agent	Mean Success Rate	Mean Timesteps

Agent 13 - Model V	96.9	13.578
Agent 11 - U*	99.9667	11.5763

Important Code Snippets

final.ipynb: Main code which has the model and its simulation.
Layer: The abstract class inherited by the activation and dense fully connected classes/objects.
FCLayer: This class creates the layer object that has the actual data on the nodes of the layer that fully connect to those of another layer with some weights and biases.
ActivationLayer: This class is just used to realize the running of the activation function for forward propagation and activation prime for backpropagation.
tanh()/lin(): Activation functions used for forward propagation in the network.
tanh_prime()/lin_prime(): Prime of Activation functions used for backpropagation in the network.
mse()/mse_prime(): Loss functions used in forward propagation and backpropagation of loss in the network.
Network: Class that represents a network object (where there are functions to add layers, add loss functions, train the network, and finally perform predictions).
Agent.move_model(): The function used for movement logic for the agent that uses Model V.
Agent.get_policy_model(): The function that actually decides the action policy based on the utility model V predicts.

The step-by-step flow for the model V with markdowns could be found in final.ipynb.

Agent 12 - $U_{Partial}$

Description

In the **partial prey information,** the state of the agent is represented by the position of the agent, vector of probabilities p (i.e the list of beliefs we used to calculate that had the probability of prey at every node in the graph) and the position of the predator.
We define the neighboring states as the state with position of the agent, the updated prey position probabilities and the position of the predator at the next timestep.
We update the prey position probabilities in accordance with project 2. (i.e multiply them with the prey_transition matrix).
The $U_{Partial}$ based agent uses a lookup against the database of optimal utilities to return a utility in the partial prey information setting. The utility of a particular state is then calculated using the formula mentioned in the write up.

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

The state policy in the partial prey information is calculated in a similar manner as in U*.

We assign utilities to each state using the formula above. At every state we calculate the utility by comparing the utilities of neighboring states the agent can transition to depending on the actions it has available at that state. The probability of transitioning to a neighboring state is calculated using the transitioning probability of predator(only) moving to the new position in the neighboring state. The action that gives us the minimum utility is state policy for that state.
We calculate the optimal state utility for every state using the **value iteration** method to solve the bellman equations (i.e non-linear system of equations) for this MDP. This in turn helps us to find optimal state policy for every state.

Q: Simulate an agent based on $U_{Partial}$, in the partial prey info environment case from Project 2, using the values of U* from above.

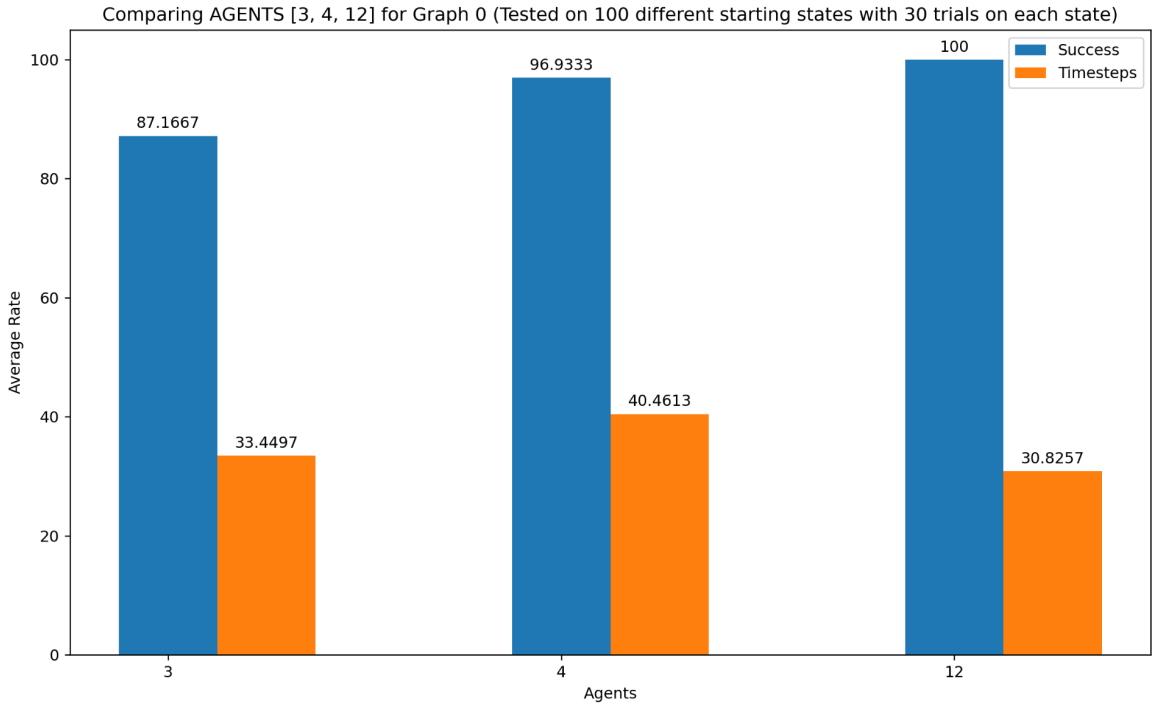
Simulated an agent by making changes to the agent.py and adding a few functions in final.ipynb to make sure the agent moves according to the description mentioned above.

Important Code Snippets

agent.Agent.get_policy(): This method calculates the next position the agent will move to according to the $U_{partial}$ strategy mentioned above.
agent.Agent.move_partial(): The method that performs the entire movement logic of this particular agent.

Q: How does it compare to Agent 3 and 4 from Project 2?

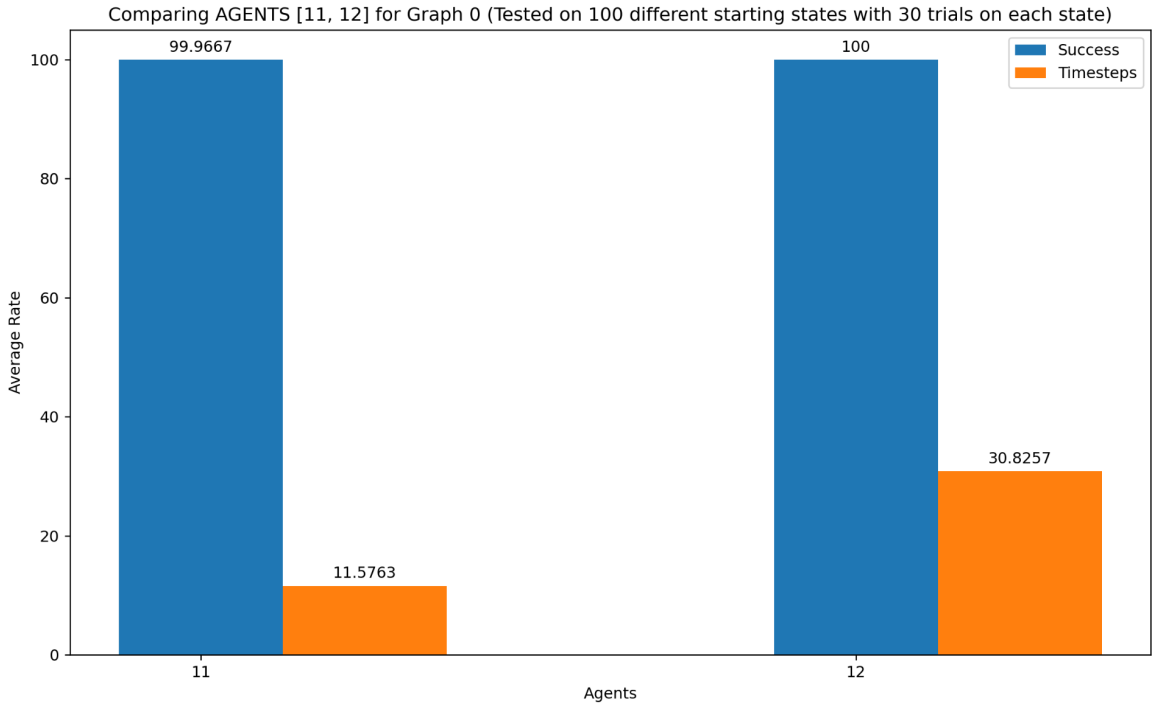
We generally see that our Agent 12 outperforms Agents 3 and 4 from project 2 in the partial prey setting as can be seen below.



Agent	Mean Success Rate	Mean Timesteps
Agent 3 - Partial Prey	87.1667	33.4497
Agent 4 - Partial Prey	96.9333	40.4613
Agent 11 - $U_{Partial}$	100	30.8257

Q: Do you think this partial information agent is optimal?

No, this partial information agent is not optimal although the success rate of agent in partial information is 100% it is possible because the cases in which we were facing failures during agent 11 U* was because the predator wasn't distracted but there is a 40% chance that it may be distracted and that may have occurred during the implementation of Agent $U_{partial}$.
If we compare the average timesteps for both the agents we can clearly see that Agent U* catches the prey in drastically less number of timesteps as compared to Agent $U_{partial}$.



Agent	Mean Success Rate	Mean Timesteps
Agent 11 - U*	99.9667	11.5763
Agent 12 - $U_{Partial}$	100	30.8257

Agent 14 - $V_{Partial}$

Description

The Model $V_{Partial}$ based agent uses a prediction logic rather than a lookup against the database of optimal utilities to return a utility in the partial prey information setting, just like $U_{Partial}$.

Q: How do you represent the states s_{Agent}, $s_{Predator}$, p as input for your model? What kind of features might be relevant?

As mentioned in the section for the model V, we tried to go ahead with distances instead of positions/belief vectors for the input features to the new model. An input constitutes a 51 feature vector with the first 50 features being scaled distances from the agent to possible prey positions and the last (51^{st}) feature being the distance to the predator.

Q: What kind of model are you taking $V_{Partial}$ to be? How do you train it?

The model we have taken V to be is a Neural Network.

For the training, we do the following:

3. The NN takes 51 inputs for every training data point (distance vector for the prey distances at every possible point (50) and distance to the predator) and derives an output based on forward propagation through activation functions on the dot products of vectors of inputs and weights and some biases.
4. It then finds the squared error (i.e. Loss) of the prediction and the training output and backpropagates it using stochastic gradient descent to update all the weights and biases at each layer.

Few peculiarities about the training:

5. The training data is standardized to get better convergence to output values.
6. The learning rate (or step size) is decayed for every data point in a batch by some decaying factor. And to avoid biases based on data ordering, the training data points in each batch are randomly ordered.
7. The model consists of 51 nodes in the input layer, followed by 102 nodes in one hidden layer and 51 nodes in another, and finally an output layer with one node.
8. What we did end up using is a tanh activation for the hidden layers and a linear activation on the output layer.

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

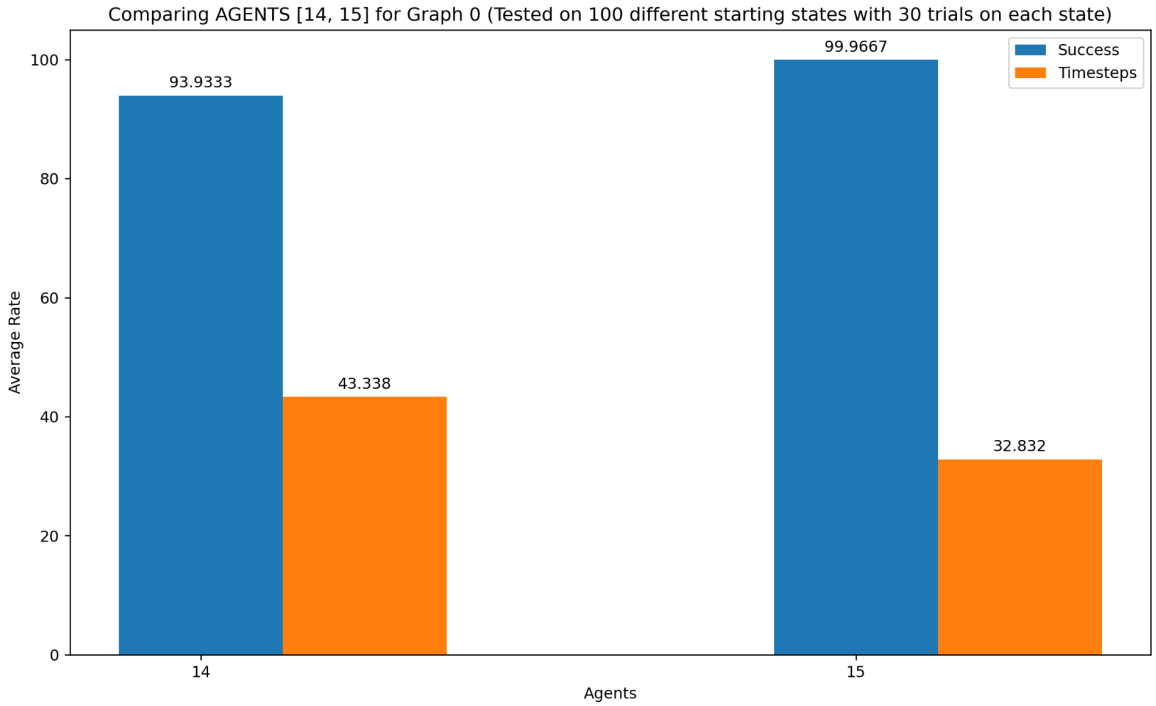
Overfitting does not seem to be an issue here as we have close to infinite states (with different corresponding utilities).

Q: How accurate is $V_{Partial}$? How can you judge this?

The model makes predictions pretty accurately, with an average error of 0.481525 (even for a small number of epochs). Also, the success rate of the agent is good without compromising on the average number of timesteps taken by the agent to catch the prey (or for the game to end) compared to the $U_{Partial}$ agent - indicating the model has not overfit to running from the predator.

Q: Is $V_{Partial}$ more or less accurate than simply substituting V into equation (1)?

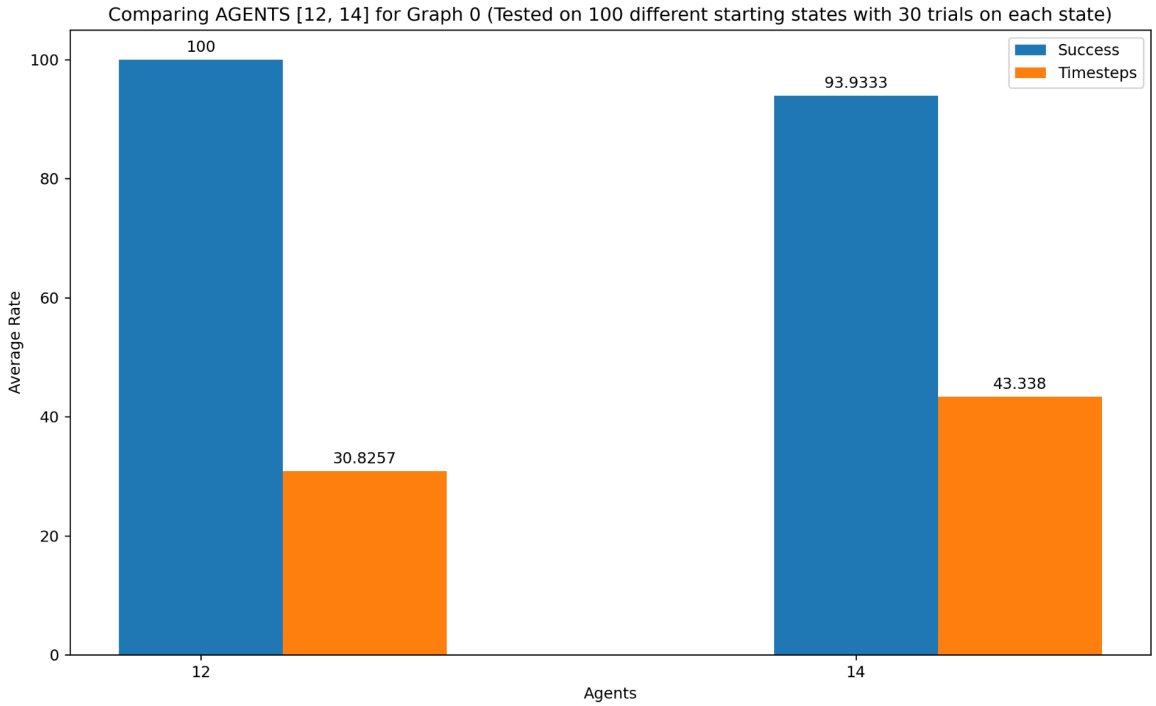
Let the substitution lead to $V_{Partial}$' or Agent 15. $V_{Partial}$ seems to be doing slightly worse than $V_{Partial}$', but that could be attributed to possible underfitting for the $V_{Partial}$ model given it was not trained enough. It still does a relatively good job catching the prey while avoiding the predator without a very significant increase in the number of timesteps. Given more training, the model $V_{Partial}$ could do better converging to a near 100% success rate.



Agent	Mean Success Rate	Mean Timesteps
Agent 14 - $V_{Partial}$	93.9333	43.338
Agent 15 - $V_{Partial}$'	99.9667	32.832

Q: Simulate an agent based on $V_{Partial}$. How does it stack against the $U_{Partial}$ agent?

In general the model does very well but does *slightly* worse compared to $U_{Partial}$. Having said that, it shows how the model already learns pretty well without a concrete formula to end up with success rates and average timesteps very close to $U_{Partial}$ for an infinite number of states and very little training.



Agent	Mean Success Rate	Mean Timesteps
Agent 14 - $V_{Partial}$	93.9333	43.338
Agent 12 - $U_{Partial}$	100	30.8257

Important Code Snippets

final.ipynb: Main code which has the model and its simulation.
Layer: The abstract class inherited by the activation and dense fully connected classes/objects.
FCLayer: This class creates the layer object that has the actual data on the nodes of the layer that fully connect to those of another layer with some weights and biases.
ActivationLayer: This class is just used to realize the running of the activation function for forward propagation and activation prime for backpropagation.
tanh()/lin(): Activation functions used for forward propagation in the network.
tanh_prime()/lin_prime(): Prime of Activation functions used for backpropagation in the network.
mse()/mse_prime(): Loss functions used in forward propagation and backpropagation of loss in the network.
Network: Class that represents a network object (where there are functions to add layers, add loss functions, train the network, and finally perform predictions).

Agent.move_partial(): The function used for movement logic for the agent that uses either of the models $V_{Partial}$ or $V_{Partial}$'.
Agent.get_policy_partial(): The function that actually decides the action policy based on the utility model $V_{Partial}$ predicts.
Agent.get_policy(): The function that actually is called by Agent 12 as well to calculate (based on equation(1)) the action policy for $U_{Partial}$, but this time based on the utility model V predicts instead of the lookup to U* in Agent 12.

The step-by-step flow for the models $V_{Partial}$ or $V_{Partial}$' with markdowns could be found in final.ipynb.

Drive Links (PKL files)

Transition Model
Transition_Model.pkl.zip
Optimal_state_utility
Optimal_su.pkl
Partial_state_utility
partial_su.pkl.zip
Agent_csv
agent_csv folder

References

Agent, Prey, Predator, Graph Utilities, Data Plotting (from project 2).

Value Iteration (to understand possible implementations)

Dr. Cowan's Lectures on Neural Networks (to understand the backpropagation logic).

The Independent Code | Neural Network (to understand how to build activation and dense layers using object-oriented code from scratch).

Improve Model Stability through Data Scaling (to understand good pre-processing practices for input data vectors)

Understanding Effects of Learning Rate (to understand better through a keras example the effects of things like a decay factor and a relevant distribution on learning rate)