CS 520 : Introduction to Artificial Intelligence

Project 3: Better Smarter Faster

RUTGERS
UNIVERSITY

TEAM MEMBERS:
Naga Vamsi Krishna Bulusu (nb847)
Anjali Menghwani (am3370)

Summary:

- ➔ This project focuses on building an agent that catches the prey as efficiently as possible using reinforcement learning.
- ➔ Given a circular undirected graph of 50 nodes with random edges with predator, prey and agent (State Space), we find the optimal utility (U*) of each state using dynamic programming algorithms and compare their performance in complete/Partial information settings.
- ➔ We also build a model to automate the process of computing optimal utility of any given state, similarly comparing the performance of the model with source truth.

Environment:

- ➔ The environment is still a circular undirected graph of 50 nodes with random, forward or backward edges along the primary loop.
- ➔ We make sure that there are no nodes that have degree more than 3 and each random edge is connected within a step size of 5 in the environment graph.
- ➔ We also spawn the agent, prey and the predator at random nodes such that the agent and predator or the agent and the prey do not spawn at the same nodes.
- ➔ We are also precomputing the shortest paths from each pair of nodes using Breadth First Search algorithm.

Prey:

- ➔ The prey is spawned at a random node which is not the same as the agent.
- ➔ The movement of prey is stochastic; it can stay in the same node or move uniformly at random to one of its neighbors.

Predator:

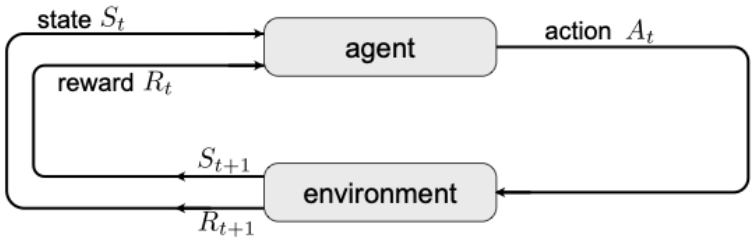
- ➔ The predator is also spawned at a random node.
- ➔ Predator follows the distracted model from P2 i.e with 0.6 probability the predator moves to one of its neighbors which are in the shortest path to the agent in t + 1 timestep and with 0.4 probability the predator moves uniformly at random to one of its neighbors.

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

- ➔ For a 50 node graph, since there are 3 actors i.e Agent, Prey, Predator.
- ➔ The total distinct states can be formulated as a counting problem with each actor occupying one of the 50 nodes.
- ➔ Therefore, total number of distinct states = 50*50*50 = 125000

U*:

- ## MDP & Value Iteration:
  - ➔ Any basic reinforcement learning is usually modeled as a Markov Decision Process (MDPs).
  - ➔ A Markov Decision Process is a 4-tuple (S, A, P_a , R_a)
    - S is set of states called State Space
    - A is a set of actions called Action Space
    - P_a (s, s′) = Pr (s_{t+1} = s′ | s_t = a, a_t = a) is the probability that action a in state s at time t will lead to state s′ at time t + 1
    - R_a (s, s′) is the immediate reward received after transitioning from state s to state s′ due to action a



- ## Value Iteration:
  - ➔ We are using the value Iteration algorithm to calculate the optimal utility for finite states.
  - ➔ In value iteration the function **π** is not used; instead, the value of **π**(s) is calculated within V(s)

$$V^*_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + V^*_i(s')\right]$$

  - ➔ Value iteration starts at i = 0 and V_0 as a guess of the value function.
  - ➔ It then iterates, repeatedly computing V_{i+1} for all states 's', until V converges.
  - ➔ Time Complexity**:**

1. **Bellman update** - *O(S*A)* [S - State space, A - Action Space, N - Number of Nodes]
2. **Value Iteration** - $O(S^2*A)$
3. **Pre computing Shortest Paths** - $O(N^2 * BFS(Source, Destination))$

- ## Objective Function:

U*(s) = minimal expected number of rounds to catch the prey for an optimal agent

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

➔ Following states are easy to determine U* for:

- The state where the agent and the prey are at the same node, U*(S) = 0
- The state where the agent and the predator are at the same node, U*(S) = infinity

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

➔

$$U*(s) = min_{a \epsilon A(s)}[r_{s,a} + \Sigma_{s'} (P_{s,s})^a \, U*(s')]$$

- U* = Optimal Utility
- $r_{s,a}$ = reward of being in state 's' and taking action 'a'
- $(P_{s,s})^a$ = Transition Probability after action 'a'
- s' = Transition state

3. Write a program to determine U $*$ for every state s, and for each state what action the algorithm should take. Describe your algorithm in detail.

➔

Algorithm for Value Iteration:

Step 1: We initialize the U* based on 3 cases:

- U*(s) = 0 { when agent_pos == prey_pos }
- U*(s) = infinity { when agent_pos == pred_pos }
- U*(s) = shortest_distance_prey_agent {other states}

Step 2:

- Next, we start the value Iteration for all 125000 states, performing bellman updates on each transition state associated with each action
- In each Bellman update we compute the product between Transition probabilities and respective Utility of that state, finally taking the cumulative sum of all utilities

Step 3:

- Next, We update the temporary U* with the minimum utility of the action states

Step 4:

- After Performing bellman updates for all states, we calculate the error between the initial U* and temporary U*

Step 5:

- If the error is below tolerance, we break the iteration, else we assign the updated U* to initial U* and we go back to step 2

<u>Pseudocode</u>:

➔ Func Value_iteration (environment):

```
tolerance -> 0.0001 # Error acceptance
U* -> dict() #{(agent_pos, prey_pos, pred_pos) : utility}
While True:
    temp_U* -> dict()
    For state in S:
        action_value = list()
        For action in A(s):
            S' -> Get_Transition_States(action)
            min_Utility->min(bellman_update(U*,
        action, S'))
        UPDATE(temp_U*) -> reward + min_utility
    error = math.inf
    For state in S:
      error=max(error, abs(temp_U*[state] - U*[state]))
    if error <= tolerance:
        Break
    U* = temp_U*
```

➔ bellman_update(U*, action, S'):

```
Utility = 0
For s in S':
  Utility += compute_transition_probability(s, action) * U[s]
    return Utility
```

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

➔ Yes, the agent will not be able to catch the prey when the utility of the starting state is infinity.

➔ In other words when the agent and predator are next to each other and the predator has an edge connecting the neighbor of the agent.

➔ The agent will die anyway.

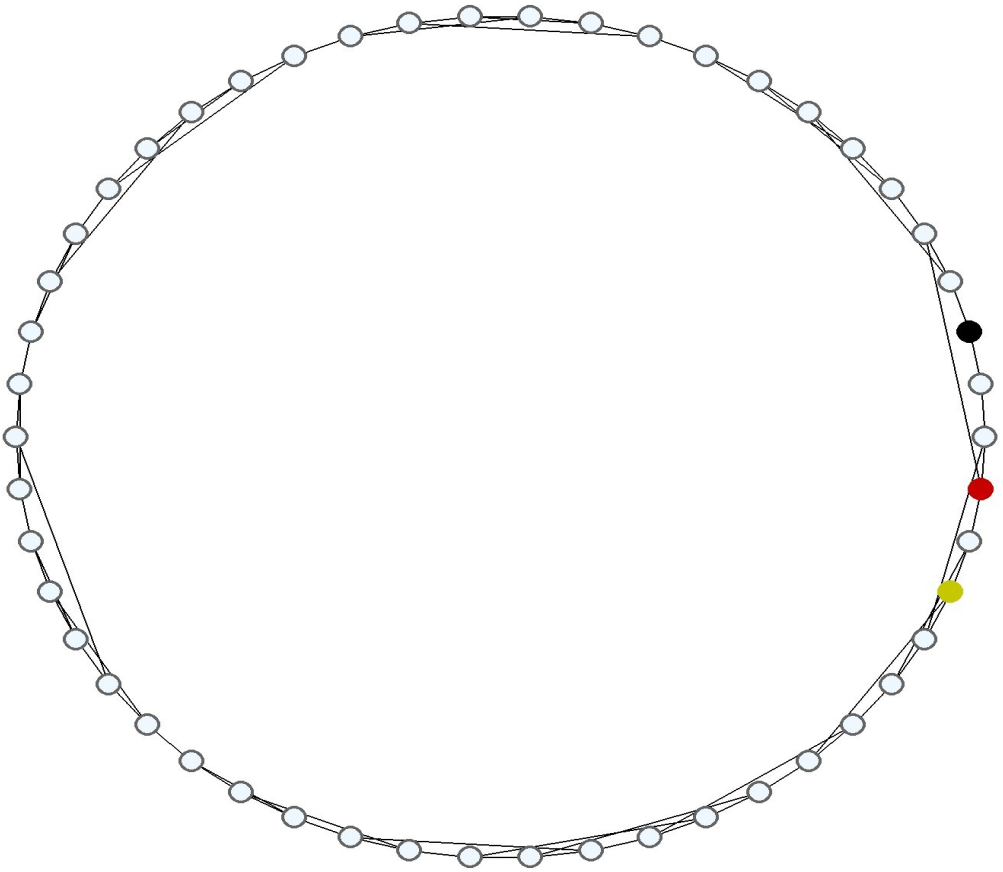
5. Find the state with the largest possible finite value of U*, and give a visualization of it.

```
lis = []
state = []
for k, v in mdp.state_ustar.items():
  if v != math.inf:
    lis.append(v)
    state.append(k)

index = lis.index(max(lis))
print('Max Finite utility:',lis[index], ', State:', state[index])
finite_max = max(lis)
```

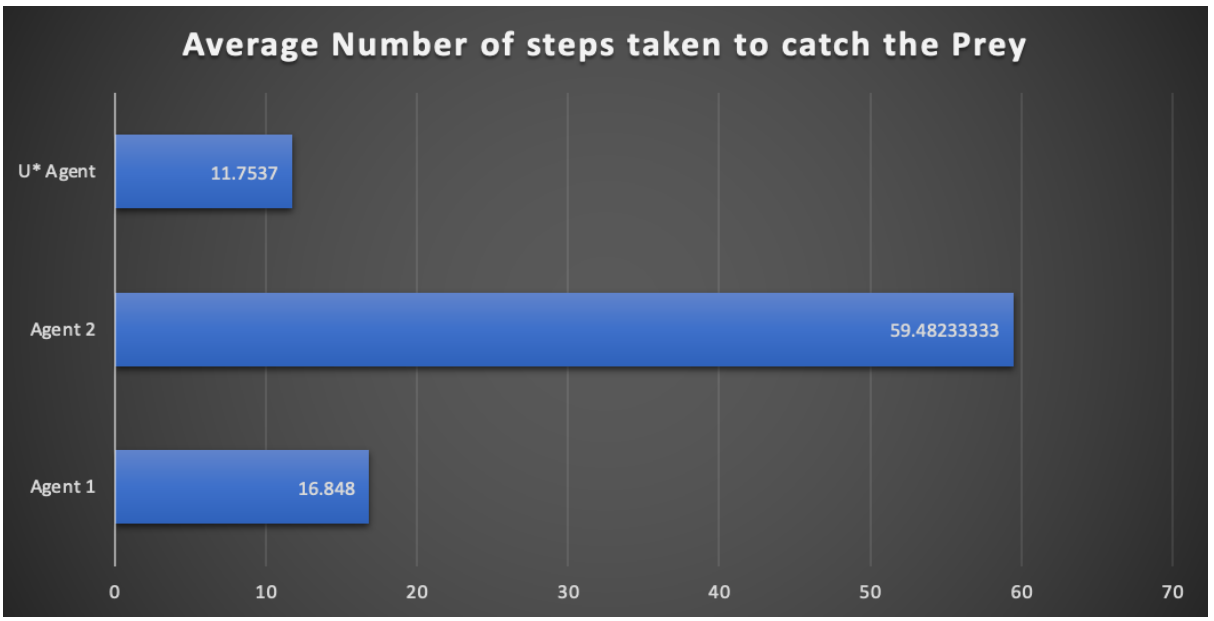
```
Max Finite utility: 26.000853854519285 , State: (48, 3, 1)
```

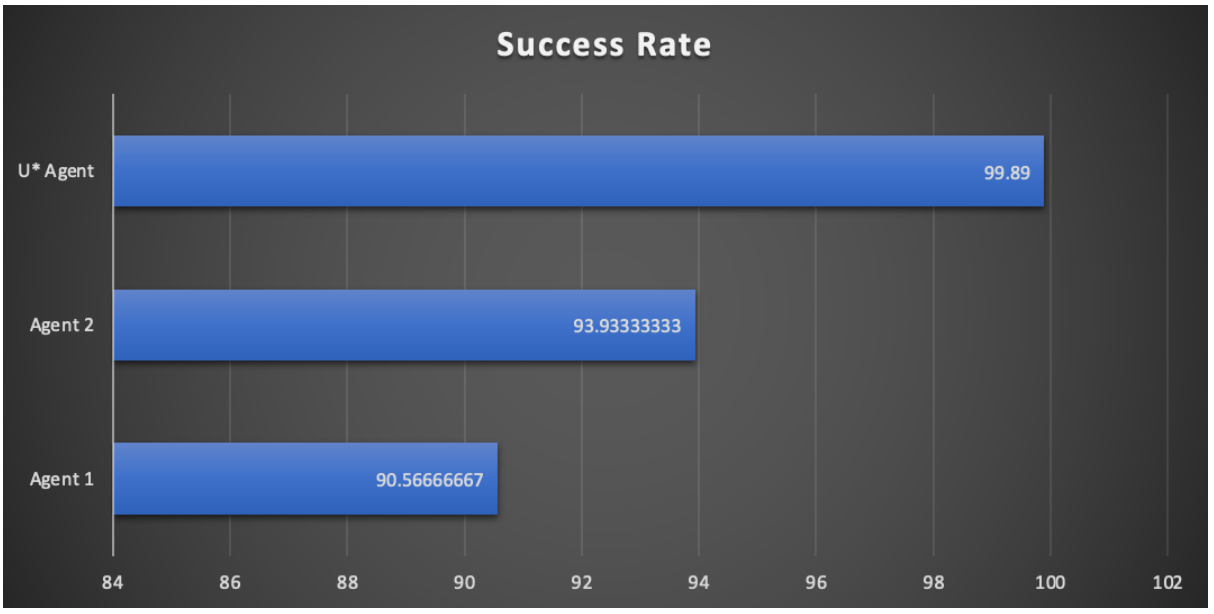
➔ **Visualization**:



Agent Prey Predator

6. **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?**





7. **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* the agent makes its choice.**
    - ➔ Yes, there is a case where U* agent and Agent 1 make different choices.
    - ➔ When prey and predator are in the same node agent1 moves toward the prey based on rule based priority.
    - ➔ Where as U* the agent looks at the utility of the states and makes a decision.
    - ➔ U* Agent and Agent 2 are almost similar in the case of survivability. Agent 2 tries to move away from the predator, where as U* search for the state with optimal utility.

V* Agent:

8. **How do you represent the states as input for your model? What kind of features might be relevant?**
    - ➔ We are representing the state 's' as input for our model as a vector. We have a total of 125000 x 5 Vector of Vectors.
    - ➔ Here, we are feeding 5 feature to our model:
        - Agent's Position
        - Prey's Position
        - Predator's Position
        - Distance between Agent and Prey
        - Distance between Agent and Predator

9. **What kind of model are you taking V to be? How do you train it?**
    - ➔ We are taking the model V to be a **Neural Network**.
    - ➔ We are training this neural network with **1 hidden layer** of 36 nodes and an output layer of 1 node.
    - ➔ In the hidden layer we use a **ReLU activation function**.
    - ➔ We have implemented **Forward Propagation** & **Backward Propagation** to train our model.
    - ➔ Forward Propagation: Here, we input the feature vectors to the neural network, via the input layer which then passes through the hidden layer in a fully connected fashion. Then the output of the hidden layer is passed through the activation function i.e. Rectified Linear Unit function. In doing so, we associate the weights which are initialized randomly at the beginning and we add biases to our input feature vectors.
    - ➔ **ReLU Activation Function:**

```
def relu(self,A1):
        return np.maximum(A1,0)
```

    - ➔ Once we get the output from these layers, we compare them with the **ground truth values**, that is, the values we get from value iteration over the graph.
    - ➔ We further use their differences to update the weights in our neural net through backward propagation.
    - ➔ Backward Propagation: Here, we move in the opposite direction, that is, now we start from the output layer to the input layer. In doing so, we compute the gradient of each connection between the nodes. We use

gradient descent to minimize the error and then use this error to update the weights and biases. Once the error converges equal to or below the mentioned threshold (2 in our case), we stop the process and assume that our model has good weights and biases.

➔ During backward propagation, we use ReLU Prime Activation Function:

```
def relu_prime(self,Z1):
    return 1. * (Z1 > 0)
```

10. Is overfitting an issue here?

➔ Yes, overfitting is an issue here, since for a given environment there are a limited number of states and each of these states will have a unique value of U*.

➔ Our model might be biased towards the specific environment

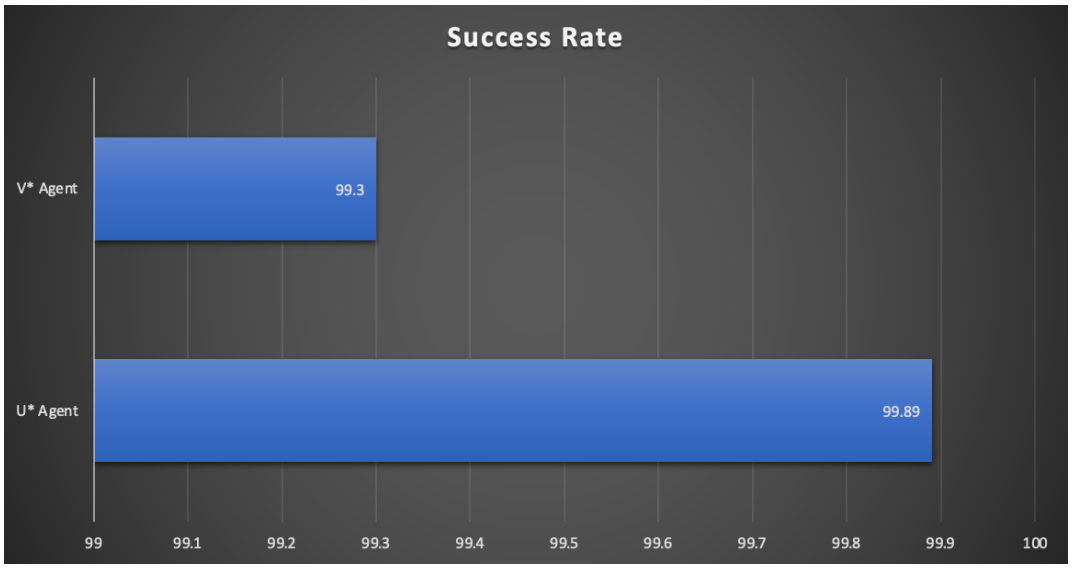
11. How accurate is V ?

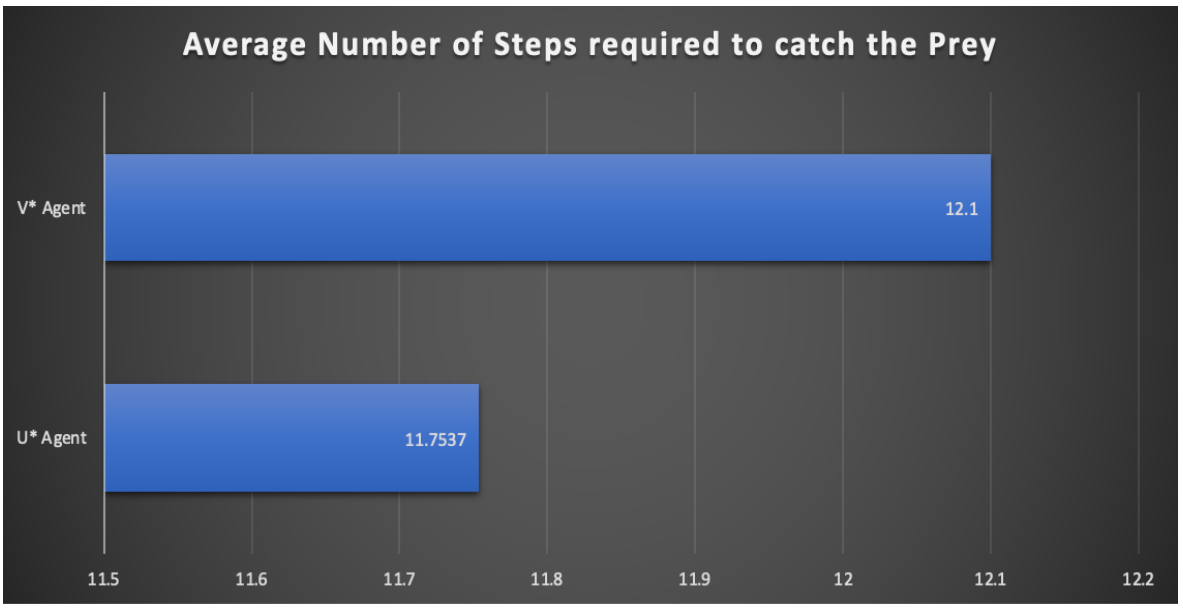
➔ Our V* model is getting a Mean Squared Error of 0.6 which resulted in predicting the values almost close to the actual U* values.

➔ Hence it performs almost same as U* with an accuracy of 99.2%

V* agent:

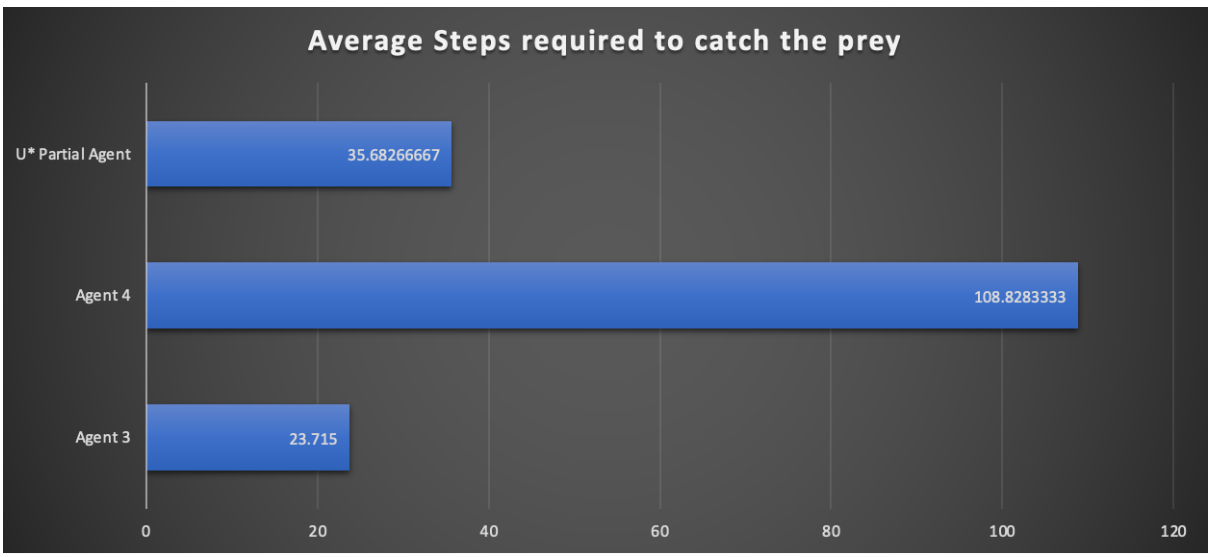
12. How does its performance stack against the U* agent?

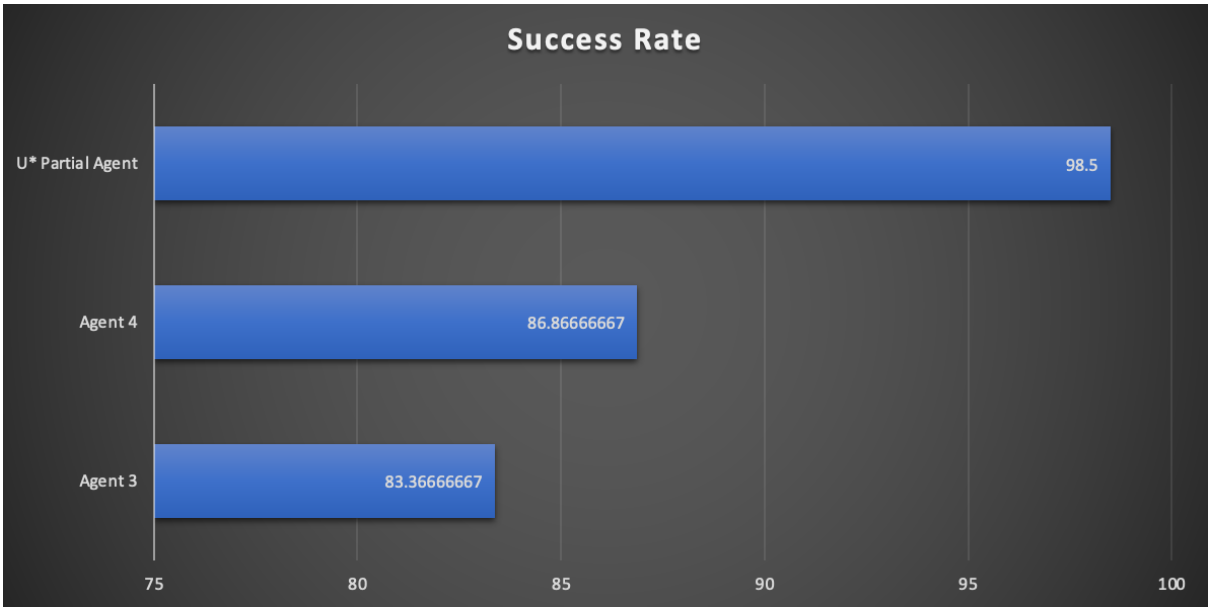




U* Partial Agent:

13. **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?**
    - ➔ It is clear from the chart that U* partial agent out performs both agent 3 and agent 4 in terms of survivability. But U* partial agent takes more steps to catch the prey when compared to agent 3.





V* Partial:

14. **How do you represent the states S_{agent}, $S_{predator}$, p as input for your model? What kind of features might be relevant?**
    - ➔ We are representing the state for our model as a vector of size 53. We have a total of 2500 Vectors of Vectors.
    - ➔ Here, we are feeding 53 features to our model:
        - Agent's Position
        - Predator's Position
        - Distance between Agent and Predator
        - 50 belief states of prey

15. **What kind of model are you taking $V_{partial}$ to be? How do you train it?**
    - ➔ We chose Neural Network as our model for Vpartial which has two hidden layers and one output layer.
    - ➔ We are passing 53 nodes for 53 features to our Input layer which consists of the agent position, predator position, the distance between agent and predator and the belief vector containing the probabilities of 50 nodes for the prey's position which is connected to the hidden layer comprising 53 nodes.
    - ➔ The input is passed through the hidden layer in a fully connected fashion.
    - ➔ The output from the first hidden layer is then dotted with a mask(a vector, the same size as the first hidden layer which has random ones and zeros) to induce dropout. By doing this we are randomly switching off a few nodes to reduce overfitting.
    - ➔ The output from the dropout is passed into the ReLU activation function

- ➔ The output from the activation function is then passed through the next hidden layer which has 106 nodes. Similar to the previous layers, dropout and activation functions are applied.
- ➔ Since we are expected to predict the floating point value which is the Upartial, the output layer comprises a single neuron which is the value of Upartial for a state s.
- ➔ We are not using an activation function for the output layer but we are just returning the linear value.
- ➔ Along With this, we are also using a bias to help shift the activation function towards the positive or negative side and hence offset the output.
- ➔ We are training the model through backpropagation similar to V* along with L2 regularization to adjust the Loss function in a way that it prevents overfitting as well as underfitting.

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

- ➔ Yes, Overfitting is an issue here, as we have a finite state of 2500 on which we can train the model, But we can encounter an infinite number of states during prediction.
- ➔ So to avoid this the model must be able to generalize to any given state unlike the V* model.

17. How accurate is $V_{partial}$? How can you judge this?

- ➔ Our V* partial model is getting a Mean Squared Error of 0.9 with an accuracy of 87.5%

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

- ➔ It will not perform very well since it has a finite set of states which cannot help in generalizing the simulation.
- ➔ We expect the performance to be very bad unlike the current implementation of V* partial.

19. Simulate an agent based on $V_{partial}$, How does it stack against the Upartial agent?