RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

Introduction to Artificial Intelligence CS 520

Project 3: Better, Smarter, Faster

ABSTRACT

This project is a continuation of the previous project (*Project 2: Circle of Life*), which is a probabilistic decision-based cat-and-mouse game. An update in this project is that rather than evaluating the best decision for the agent based on uncertain probabilistic information (Bayesian Networks), we use Reinforcement Learning (Utility) to give the optimum success rate for the agent. There are four models for the agent –

U* Agent and V Agent

U_partial Agent and V_partial Agent

The Three Entities:

The three entities in this game are the same as the previous project –

The Agent

The Prey

The Predator

The basic cat-and-mouse chase concept of these three entities is that the agent is pursuing the prey while simultaneously being chased by the predator. The agent must catch the prey while avoiding getting captured by the predator chasing him.

End States:

There are 3 end states for this game.

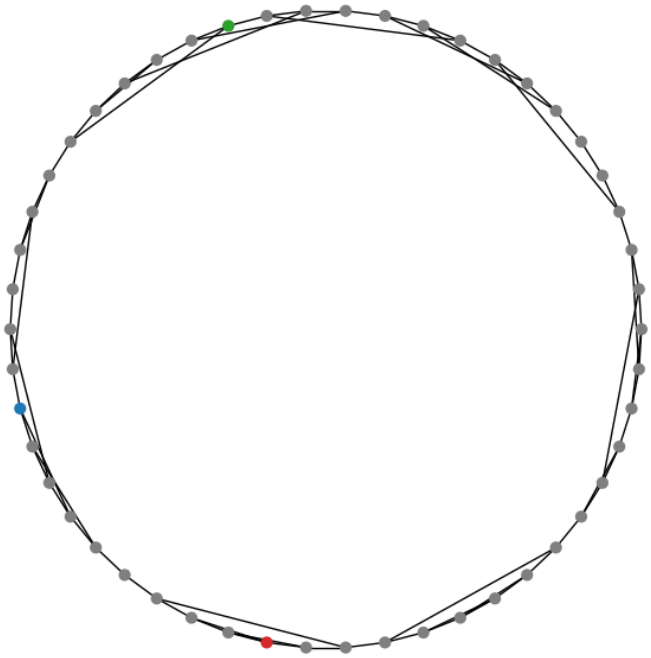
End State 1: The agent occupies the node containing the prey. This denotes as the agent won the game.

End State 2: The agent occupies the node containing both the prey and the predator. This denotes a "game over" where nobody would win.

End State 3: The predator occupies the node containing the agent. This denotes as the agent lost and the predator won the game.

ENVIRONMENT

The environment is the same one as the previous project 2. We have an undirected graph length of 50 nodes. Fifty nodes form a large circle numbered 1 to 50. This graph can also include additional edges that connect one node to another in five backward or forward steps. The degree of each node should be at most 3. Any node in this environment cannot have an edge over itself. The agent will not spawn in any game-ending state. The prey and predator can spawn on the same node and co-exist within the same node without any issue. In the figure shown, the **GREEN** node represents the prey, the **RED** represents the predator, and the **BLUE** node represents the agent.



The Cycle of Entities:

The agent always takes the first step, starting off the game, followed by prey and predator.

Prey:

The prey chooses among its node neighbors, including its own position (staying where it is), and jumps to any of them with a defined probability of ¼ (0.25) (degree = 3) or 1/3 (0.333) (degree = 2) in the environment

Predator:

Here, the easily distracted predator chooses a neighboring node with the shortest distance to the agent with a 0.6 probability. The predator may choose the other neighbors with a probability of 0.4, which may not lead directly to the agent randomly.

(Below is the sample code for environment generation, We used NetworkX Library for environment)

```
def create_graph():
    all_random_nodes = []
    G = nx.Graph()
    G.add_nodes_from(range(1, 51))
    G.add_edges_from([(1, 2), (1, 50)])
    for node in range(2, 51):
        if node == 50:
            G.add_edges_from([(node, node - 1), (node, 1)])
            continue
        G.add_edges_from([(node, node + 1), (node, node - 1)])

    while len(all_random_nodes) != 50:
        random_node_r_neigh = []
        random_node_l_neigh = []
        random_node_neighs = []

        random_node = random.choice(list(G.nodes))
        if random_node in all_random_nodes:
```

```
            continue
        all_random_nodes.append(random_node)

        if G.degree[random_node] == 2:
            n = 2
            last_element = 50
            first_element = 1
            while n <= 5:
                if (random_node - n) > 0:
                    random_node_l_neigh.append(random_node - n)
                else:
                    random_node_l_neigh.append(abs((random_node - n) +
last_element))

                if (random_node + n) <= 50:
                    random_node_r_neigh.append(random_node + n)
                else:
                    random_node_r_neigh.append(abs((random_node + n) -
last_element))
                n = n + 1

            random_node_neighs = random_node_l_neigh + random_node_r_neigh

            for i in range(len(random_node_neighs)):
                attach_node_rand = random.choice(random_node_neighs)
                if G.degree[attach_node_rand] != 2:
                    continue
                G.add_edge(random_node, attach_node_rand)
                G.add_edge(attach_node_rand, random_node)
                break
        else:
            continue

    return G


G = create_graph()
begin_agent_pos = 0
begin_prey_pos = 0
begin_predator_pos = 0

agent_random_loc = random.choice(list(G.nodes))

while True:
    prey_random_loc = random.choice(list(G.nodes))
    predator_random_loc = random.choice(list(G.nodes))
    if prey_random_loc == agent_random_loc or predator_random_loc ==
agent_random_loc:
        continue
    break

agent_pos = agent_random_loc
prey_pos = prey_random_loc
predator_pos = predator_random_loc

print('agent_pos = ', agent_pos)
print('prey_pos = ', prey_pos)
print('predator_pos = ', predator_pos)

# DRAW GRAPH
color_map = []
```

```
for n in list(G.nodes):
    if n == agent_pos:
        color_map.append('tab:blue')
        continue
    if n == prey_pos:
        color_map.append('tab:green')
        continue
    if n == predator_pos:
        color_map.append('tab:red')
        continue
    color_map.append("tab:gray")

pos = nx.circular_layout(G)
# node_size=10
nx.draw(G, with_labels=False, pos=pos, node_color=color_map, node_size=40)
plt.show()
```

AGENT MODELS

Algorithm to determine U ∗ for every state (s), and for each state, what action the algorithm should take

To calculate the U* for every state (s), we used Bellman Equation with the value function to find the best optimal value for every state. Unlike the policy function, the value function calculation for Bellman Equation for a state (s) can be calculated using the formula:

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

This equation denotes the reward the agent got upon leaving a state plus the discounted value of the state we landed multiplied by the transition probability of the agent moving into that state. We used a dynamic programming value iteration function to calculate the best utilities based on the previous utilities. The limiters for the U* values for every state are

- '0' if the agent and the prey share the same node
- 'Infinity' if the agent and the predator share the same node
- '1' as a reward for each step of the agent moving towards the prey

These are the beginning U* values, which serve as the basis for the following utility values of other states. We used *math.inf* for the infinity value utility update. We iterate over each possible next state from the current state and calculate the utilities of those states using Bellman's Equation value iteration function. We calculate the utilities for the entire possible future states and assess the exact difference between the current and previous utility values for that particular state.

(Below is the sample code)

```
rewd = 1 # 0.1 Always Total
dis_penl = 0.0002

ct = 0
for current_agent_pos in range(50):
    for current_prey_pos in range(50):
        for current_predator_pos in range(50):
```

```
            u_str[(current_agent_pos, current_prey_pos,
current_predator_pos)] = random.randrange(5, 35)
            ct = ct + 1

# Using Value dis_itr
for current_agent_pos in range(50):
    for current_prey_pos in range(50):
        for current_predator_pos in range(50):
            if current_agent_pos != current_predator_pos and
current_agent_pos == current_prey_pos:
                u_str[(current_agent_pos, current_prey_pos,
current_predator_pos)] = float(0.)

            elif abs(current_prey_pos - current_agent_pos) == 1:
                u_str[(current_agent_pos, current_prey_pos,
current_predator_pos)] = 1.0

            elif current_agent_pos == current_predator_pos:
                u_str[(current_agent_pos, current_prey_pos,
current_predator_pos)] = math.inf

            else:
                u_str[(current_agent_pos, current_prey_pos,
current_predator_pos)] = rewd
# Here I am using val_itr
u_str = u_str
while x:
    act = []
    err_max = 0
    util_actv.clear()
    for current_agent_pos in range(50):
        for current_prey_pos in range(50):
            for current_predator_pos in range(50):
                if current_agent_pos != current_predator_pos \
                        and current_agent_pos == current_prey_pos:
                    util_actv[(current_agent_pos, current_prey_pos,
current_predator_pos)] = 0.0

                elif abs(current_prey_pos - current_agent_pos) == 1:
                    util_actv[(current_agent_pos, current_prey_pos,
current_predator_pos)] = 1

                elif abs(current_agent_pos - current_predator_pos) \
                        or current_agent_pos == current_predator_pos == 1:
                    util_actv[(current_agent_pos, current_prey_pos,
current_predator_pos)] = (10 * (10 ** 8))

                else:
                    for agd in main_graph[current_agent_pos]:
                        nei_prd_pos = []
                        nei_py_pos = [current_prey_pos]
                        nei_py_pos.extend(main_graph[current_prey_pos])

nei_prd_pos.extend(main_graph[current_predator_pos])
                        # Evaluation
                        fut_util = 0
                        probs_tot = 0
                        for py in nei_py_pos:
                            for prd in nei_prd_pos:
                                dtt_bfs = BFS(current_agent_pos,
current_predator_pos)
```

```
                                sorted_dtt_vals = []
                                for dt1 in sorted(dtt_bfs,
                                                  key=lambda k:
len(dtt_bfs[k]), reverse=False):
                                    sorted_dtt_vals.append(dt1)
                                dt2 = len(sorted_dtt_vals)
                                xg = random.randrange(1, 4)
                                dtt = ((xg * 0.6)/dt2) +
(0.4/len(main_graph[current_predator_pos]))
                                probs = ((1.0 / len(nei_py_pos)) * dtt)
                                probs_tot = probs_tot + probs
                                fut_util = (fut_util + probs *
((u_str[(agd, py, prd)]) + rewd))
                            actv_acts = fut_util
                            act.append(actv_acts)
                        util_actv[(current_agent_pos, current_prey_pos,
current_predator_pos)] = min(act)

    for current_agent_pos in range(50):
        for current_prey_pos in range(50):
            for current_predator_pos in range(50):
                err_x = err_max, abs(util_actv[(current_agent_pos,
current_prey_pos, current_predator_pos)] -
                                     u_str[(current_agent_pos,
current_prey_pos, current_predator_pos)])
                err_max = max(err_x)

    if err_max < dis_penl:
        break

    u_str = util_actv
```

We stored a single environment's U* value data because the environment is randomly generated every time. Below is the code for storing the U* data in a system directory for future use.

```
file_to_save_env = open('C:\Users\Arthur
King\PycharmProjects\520_INTROAI_P3\Env', 'wb')
pkl.dump(main_graph, file_to_save_env)
file_to_save_env.close()
file_to_save_u_vals = open('C:\Users\Arthur
King\PycharmProjects\520_INTROAI_P3\u_vals', 'wb')
pkl.dump(u_str, file_to_save_u_vals)
file_to_save_u_vals.close()
```

Build a model to predict the value of U ∗ (s) from the state s. Call this model V.

For this model V, we are using an Artificial Neural Network for predicting the values of U* that performs on three layers. We are using five features on our input layer, which include the positions of the agent, predator, and prey, as well as the distances between them. This input layer is connected to another layer, which comprises 8 neurons in the AN network. The model V* has no alternative way to do better than the simulation because it can only learn to get near to the values of U*. Therefore, V* never outperforms U* and always performs at least as well as U*. Model V has an accuracy of 96.18%e when we trained it using U* data. Gradient descent and backpropagation are used in this process. In order to update the values of the weights and biases in accordance with our learning rate, gradient descent is utilized to calculate the gradients of the weights and biases. we

tested the model with varying tolerance and learning rates and came up with a good threshold to output maximum accuracy.

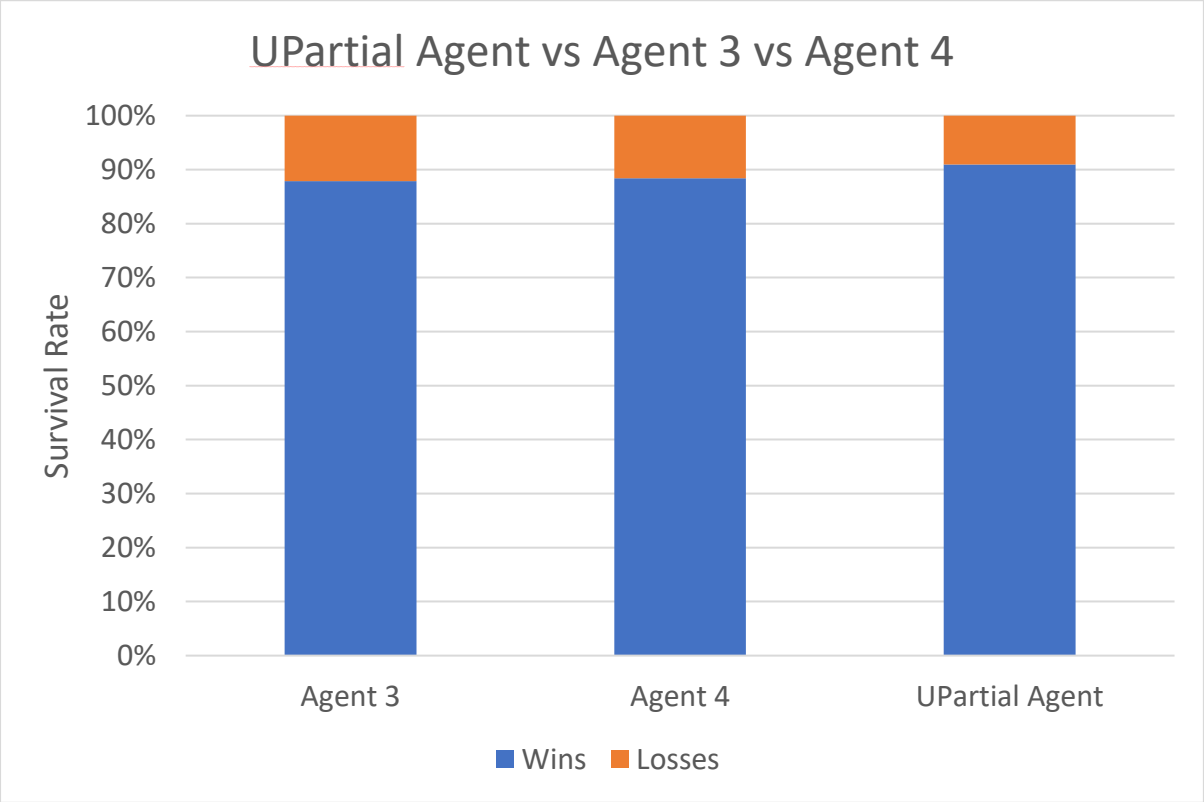
(Sample Code)

```
class Network:

    def __init__(self, x2, x1):
        self.sets_t = None
        self.ND_1 = np.random.randn(36, 5) * 0.01
        self.ND_2 = np.random.randn(1, 36) * 0.01
        self.n_1 = np.zeros((36, 1))
        self.n_2 = np.zeros((1, 1))
        self.x2 = np.transpose(x2)
        self.x1 = np.transpose(x1)

    def run_func(self):
        l1 = np.mean(self.x1 - self.sets_t)
        loss = ((l1 ** 2) * 0.5)
        # print('Loss = ', loss)
        print('Am I running?') # Checker

    def tol_tot_main(self, phi_1):
        phi_1[phi_1 >= 0] = 1
        phi_1[phi_1 < 0] = 0.01
        return phi_1

    def tol(self, all_sets_list):
        return np.maximum(all_sets_list, 0)

    def tol_tot(self, all_sets_list):
        all_sets_list[all_sets_list < 0] = 1 * 0.01 *
all_sets_list[all_sets_list < 0]
        return all_sets_list

    def tol_main(self, phi_1):
        return (phi_1 > 0) * 1 * 1.0

    def forth(self):
        self.phi_1 = np.dot(self.W1, self.x2) + self.b1
        self.all_sets_list = self.tol(self.phi_1)
        self.phi_2 = np.dot(self.W2, self.all_sets_list) + self.b2
        self.phi_2 = self.phi_2

    def reverse(self):
        mn = self.x1.shape[1]
        entei_2 = (self.sets_t - self.x1)
        dum_2 = (1 / mn) * (np.dot(entei_2, self.all_sets_list.T))
        sm_2 = (1 / mn) * np.sum(entei_2)
        entei_1 = np.multiply(np.dot(self.ND_2.T, entei_2),
self.tol_main(self.phi_1))
        dum_1 = (1 / mn) * (np.dot(entei_1, self.x2.T))
        # print('Is it working ?') FIXED
        sm_1 = (1 / mn) * (np.sum(entei_1))
        self.W1 = self.ND_1 - 0.001 * dum_1
        self.b1 = self.n_1 - 0.001 * sm_1
        self.W2 = self.ND_2 - 0.001 * dum_2
        self.b2 = self.n_2 - 0.001 * sm_2
        # print('crct vals = ', dum_1)
        print('vals = ', sm_1, sm_2)
```

```
        #print('rr = ', dum2)
        return dum_1, sm_1, dum_2, sm_2
```

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?

Using the values of U*, we calculate the value for Upartial. Like in the previous project of partial information setting, we do not, however, know the precise location of the prey in this scenario. We have a belief state dictionary that stores the likelihood of the prey for each state s in the state space to address this. In a manner similar to Agents 3 and 4 from Project 2, we update the belief states following the movement of the agent and the prey.

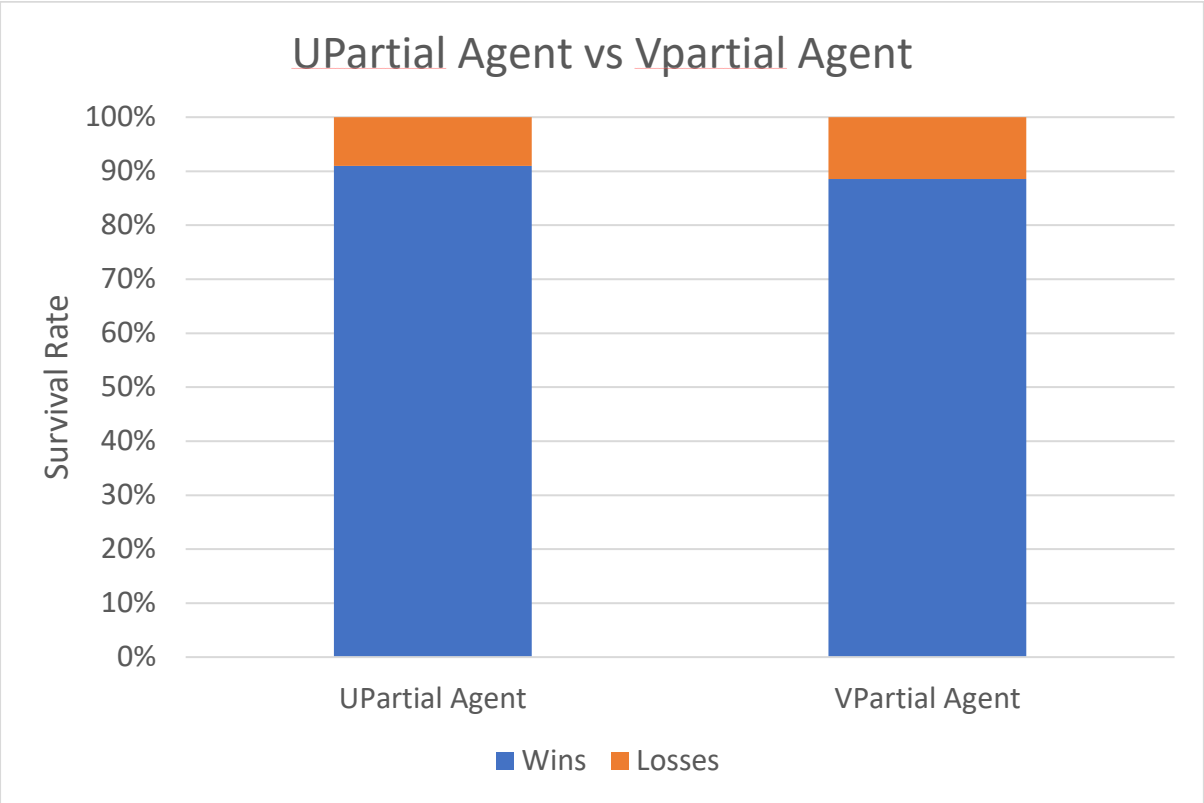


Yes, we think the UPartial agent is indeed optimal compared with the previous project agents. Not just the UPartial agent but all the agents clearly performed a lot better than all the previous agents, which corroborate optimal performance.

Build a model Vpartial to predict the value Upartial for these partial information states. Use as the training data states (including belief states) that actually occur during simulations of the Upartial agent.

The positions of the agent, the predator, the distance between them, and the belief states holding the probability of each node for the position of the prey make up the input of our model. These appear to be the best-fit features for the model. The belief states containing the probability of 50 nodes for the prey's position are connected to a layer of 38 neurons. We are passing 38 neurons for 50 features, which consist of the agent position, predator position, the distance between agent and

predator, and the belief states. This model is quite intensive compared to the others because of the increase in the number of features to fit. The survivability of this model is 87.54%



GRAY BOX QUESTIONS

(Some of the gray box questions are explained with the models themselves)

Gray Box Question 1: How many distinct states (configurations of the predator, agent, prey) are possible in this environment?

- ✓ The environment consists of 50 nodes in total. A "state" represents an arbitrary game-beginning position, where the agent, the prey, and the predator each occupy a node to start the game. According to the permutation and combination rule, we can have **50x50x50**, which equals (~)**125,000** distinct states in the environment, by taking into account 50 specific locations for the agent, prey, and predator.

Gray Box Question 2: What states (s) are easy to determine (U ∗) for?

- ✓ For a given state (s), U*(s) will always be the minimum expected number of rounds to catch the prey for the optimal agent. The total number of possible distinct states is 125,000; among those states, the possibility of the agent and the predator sharing the same node is 50 x 50 states with ~ 2500 possibilities. For these states, the U*(s) will be 'infinity-inf' because the agent would have died ending the game. Likewise, if the agent and the prey among those 125,000 possibilities share the same node ending the game, the U*(s) for these states will be '0', and there will be 2500 such possible states.

Gray Box Question 3: How does U*(s) relate to U* of other states and the actions the agent can take?

- ✓ U*(s) denotes the U* value of a specific state 's'. The U*(s) defines the optimal utility of a state s, which is the minimum number of steps taken to get to all the possible next states from the current state (s). The actions the agent can take based on the U* values of each state s will be the most optimal decisions. We can calculate the U* of every possible state using Bellman's Equation with a value function. Mathematically, we can define Bellman's Equation as

$$v(s) = \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]$$

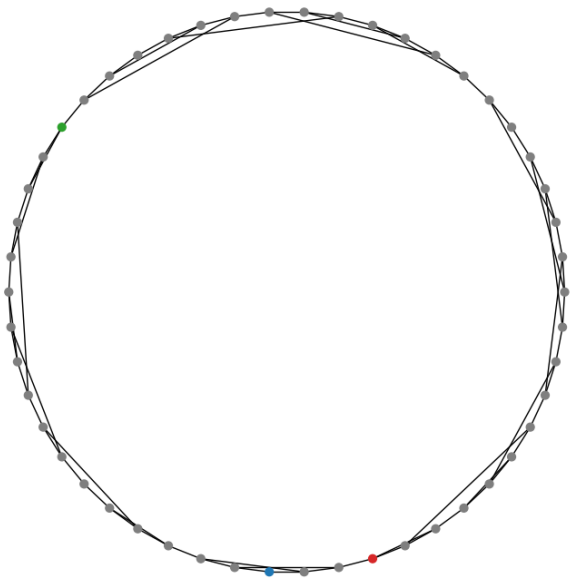
we can say how U*(s) relates to U* of other states in the following way – If the agent in some state (s) moves from that state to some other state (s'), then we can evaluate how good it was for the agent to be in that state(s) using Bellman's Equation by rewarding the agent for leaving the state(s) plus the value of the next state (s') the agent went.

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

- ✓ The agent won't be able to capture the prey when the utility value is infinity, which means that the predator and the agent are in the same node. This failure occurs only in 50 x 50 states (~ 2500). But, this sort of starting position will not be applicable in our case as the agent and the predator (or prey) will never spawn in the same node to create such a state failure.

Gray Box Question 5: Find the state with the largest possible finite value of U $*$, and give a visualization of it.

- ✓ According to the initial utility limiter setting, the largest value would be infinity, which occurs when the predator occupies a node that has the agent in it. But, to assess the state with the largest 'finite' value of U* then, we should exclude such failure states.

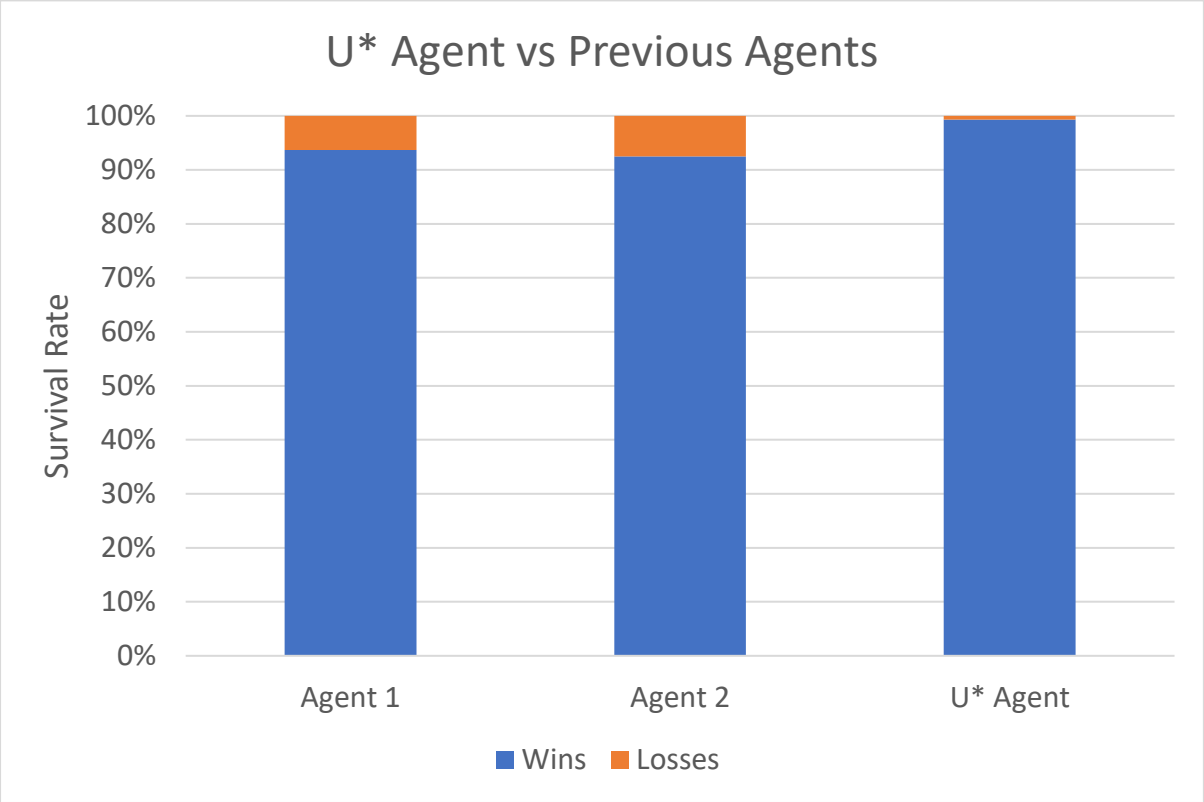


Largest Possible Finite Value of U*: 13.88 with entity node positions as
[Agent = 38, Predator = 41, Prey = 21]

As shown above, the most significant finite value we got for the U* is on a specific environment. There is also a possibility of even larger finite U* values depending upon the environment.

Gray Box Question 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?

- ✓ The initial U* agent will perform on an information setting that shows him the exact locations of the prey and the predator for each step, just like in previous project 2. Unlike the agent in the previous project, who moves according to some set rules, the U* agent is built for optimum peak performance and takes decisions/chooses nodes based on the utility values computed prior.



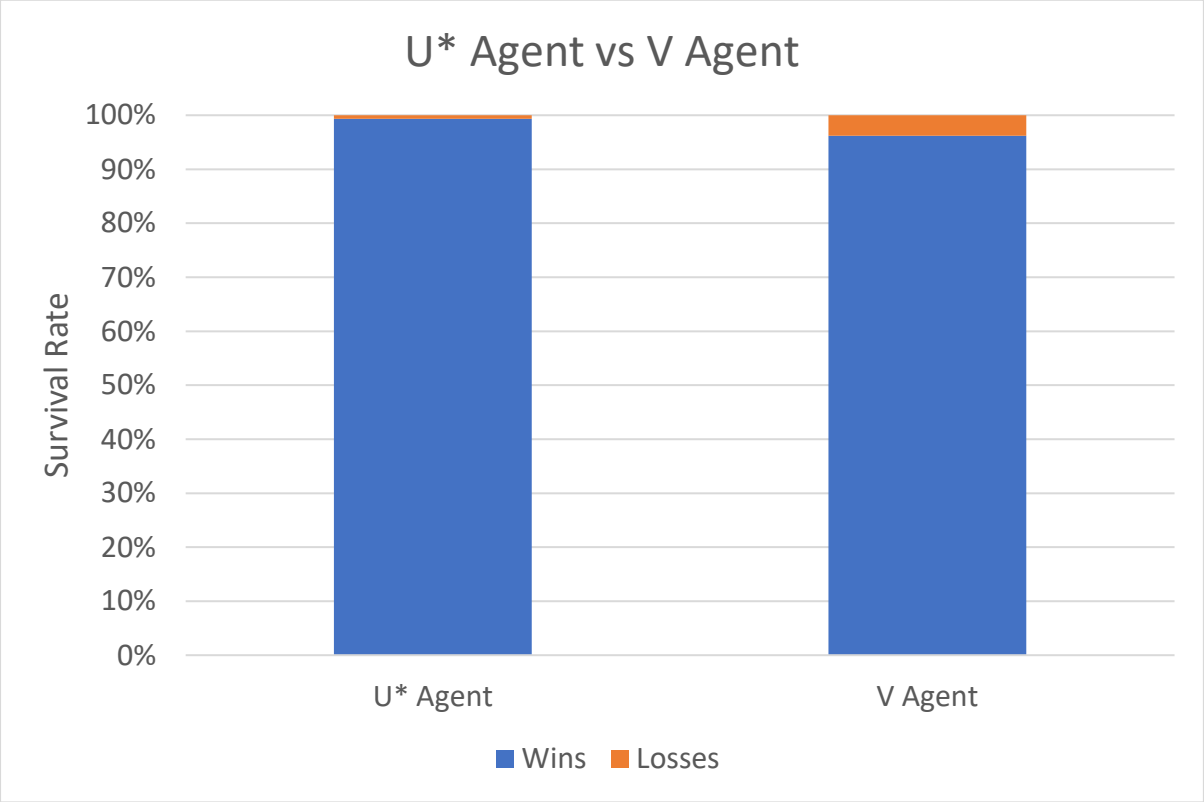
Gray Box Question 8: Are there states where the U ∗ agent and Agent 1 make different choices? The U ∗ agent and Agent 2? Visualize such a state, if one exists, and explain why the U ∗ agent makes its choice.

- ✓ Because the movement of Agents 1 and 2 from the previous project is governed by a pre-defined order of rules based on the distances between the agent, prey, and predator, they are not very efficient. Still, the U* agent will sometimes make different decisions than agents 1 and 2 based on the optimal utility value of the current setting configuration. U* agent always chooses the node not on some set rules but rather on ***maximizing the reward*** throughput in the long run. So, it is very possible for the U* agent to make different choices fundamentally.

Gray Box Question 9: How do you represent the states as input for your model? What kind of features might be relevant?

- ✓ For our model V, we found that the best relevant features for maximum accuracy are the agent, prey, and predator positions and the distances between each of them for each state.

Gray Box Question 10: How does its performance stack against the U ∗ agent?



V Model is either as efficient as U* or slightly worse than U* because of its value predictions which don't guarantee absolute values, unlike U*.

Submitted by:

Dheekshith Dev Mekala

dm1653

Sai Surya R

sr1789