ABSTRACT

CIRCLE OF LIFE:

This project is a probabilistic decision-based cat-and-mouse game. There is a level of uncertainty involved all over in this game, and the decision-making of all the entities in this game depends on that uncertainty and how those entities assess and assimilate that uncertainty. This project has four information settings that affect how the entities calculate/evaluate the uncertainty present in this game, i.e., probabilistic information.

The Three Entities:

The three entities in this game are as follows -

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 has to catch the prey while avoiding getting captured himself by the predator that is chasing him. The different settings of this game define how the agent corresponds to the environment as well as his ability to see or keep track of an entity.

The Four Information Settings:

1. The complete information setting: At every timestep in the environment, the agent knows the exact location of the prey and predator and makes decisions accordingly to catch the prey and avoid the predator.

The agent in the below settings uses a probabilistic model to assess the uncertainty of the environment i.e., assess the certainty of an entity present in a particular node.

2. The Partial Prey Information Setting: At every timestep in the environment, the agent knows the exact location of the predator but does not know the location of the prey it needs to catch. In this setting, to overcome this lack of information, the agent has the special ability to survey a node before taking a step, i.e, you can assume this survey to be equivalent to sending out a drone in advance to any random location within the environment to check whether the prey exists in that location.

3. The Partial Predator Information Setting: At every timestep in the environment, the agent knows the exact location of the prey but does not know the location of the predator it needs to avoid. The same ability to survey or send out a drone to a random node in the environment beforehand exists.

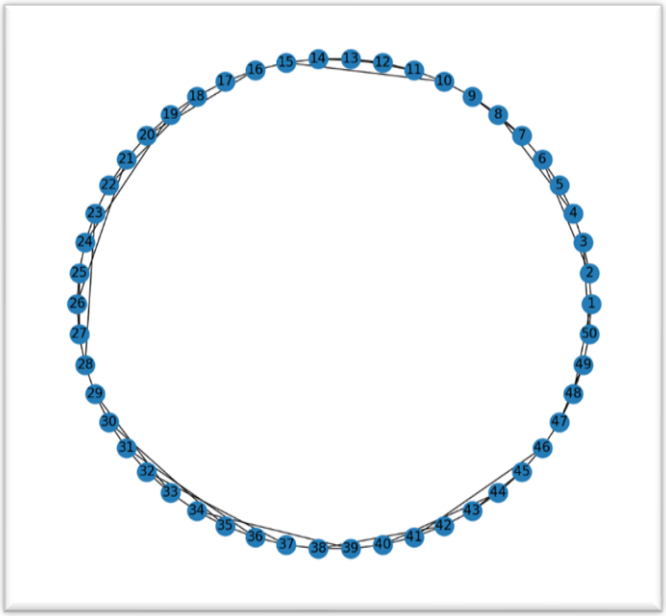
4. The Combined Partial Information Setting: In this environment setting the agent does not know both where the prey and predator are at any given timestep. The same agent's ability to survey a node is available but will only be able to survey based on the belief states of prey

or predator. The agent is completely blind in this setting but makes decisions based on the belief states of probabilities of both the prey and the predator.

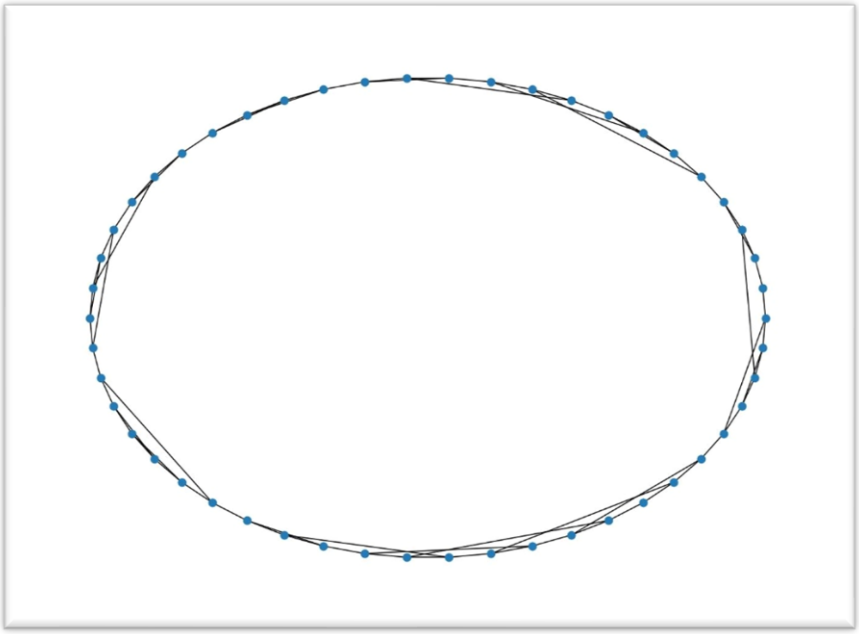
There is a concept of the base agent as well, whose decisions are affected based solely on the prey effectively ignoring the predator. The base agent's goal is to greedily catch the prey, always taking the best shortest step toward the location of the prey without heeding the movement of the predator at all.

THE ENVIRONMENT

The environment for this project is a network of nodes connected by edges. The agent, prey, and predator can all travel between nodes along the edges. 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. The additional edges to add within five steps of a node are to ensure proper traversal without randomly connecting to different nodes spread throughout the environment. Any node in this environment cannot have the edge over itself. Nodes with only two edges are possible after every other node is connected with something else, and their degree is already 3.



For the representation of the nodes in this environment, I used a dictionary to hold every node value and its connected vertices (Edges). All the beginning positions of the agent, prey, and predator are initialized to zero. After the environment is created conforming to the rules as stated in this project write-up, the actual occupying positions of the agent, prey, and predator in the nodes are arbitrarily chosen with the condition that the agent should never spawn in a pre-occupied node at the start of the game. The prey and predator can generate on the same node and can co-exist within the same node without any discrepancy.



(Below is the sample code for environment generation)

```
begin_agent_pos = 0
begin_prey_pos = 0
begin_predator_pos = 0

main_graph = {1: [2, 50]}

all_random_nodes = []
# Change 50
for node in range(2, 51):
    # Change 50
    if node == 50:
        main_graph[node] = [1, node - 1]
        continue
    main_graph[node] = [node + 1, node - 1]
# Change 50
while len(all_random_nodes) != 50:
    random_node_r_neigh = []
    random_node_l_neigh = []
    random_node_neighs = []

    random_node = random.choice(list(main_graph.keys()))
    if random_node in all_random_nodes:
        continue
    all_random_nodes.append(random_node)

    if len(main_graph[random_node]) == 2:
        n = 2
        # Change 50
        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(random_node - n + last_element)
                # last_element = last_element - 1
```

```
                # Change 50
            if (random_node + n) <= 50:
                random_node_r_neigh.append(random_node + n)
            else:
                random_node_r_neigh.append(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 len(main_graph[attach_node_rand]) != 2:
                continue
            main_graph[random_node].append(attach_node_rand)
            main_graph[attach_node_rand].append(random_node)
            break
    else:
        continue

print(main_graph)
```

Graph Theory Question:

According to the above implementation of the graph environment, the maximum number of adding additional edges conforming to the degree 3 for each node is 25. The maximum number of edges (vertices) you can add to any environment at max is half of the environment's total nodes. But the general, most probable total number of edges you can add is (half of the environment's total nodes - 1), i.e., in our scenario with the highest probability at most, 24 edges were added. We generated several other graphs to evaluate the veracity of this condition, and the least number of node connectivity for this environment was 18 edges at most (This has the least probability of occurring considering the weight distributions of the ideal environment as above). So, according to the results and adhering to the ideal environment weight distributions, the maximum probability occurrence of node connectivity is 24 edges, followed by 23 edges and 25 edges. From this data, we can ascertain that the maximum number of edges is 25 and the minimum is 18, while the most probable occurrence is 24 edges for the entire environment.

CYCLE OF THE THREE ENTITIES (AGENT, PREDATOR, PREY):

The concept "distance" for assessing the actual distance between any two nodes is Uniform-Cost Search (UCS) (Here, BFS as the cost to every node is 1 (Non-Weighted Graph))

- The movement decision of the agent varies according to the information game settings as specified. Regardless of the information setting, the agent always takes the first turn.
  - Base Information Setting: The agent's movement decisions are completely oblivious to the predator's movement while choosing nodes that are shortest towards the prey's location, greedily trying to catch the prey.
  - Complete Information Setting: The agent always makes informed movement decisions based on the locations of both the prey and the predator, effectively trying to avoid the predator and catch the prey.
  - Partial Prey Information Setting: The agent makes informed movement decisions based on the predator's location, constantly trying to avoid it while making probabilistic partial

informed decisions towards the prey's possible location. If the agent is certain of the prey's location, he will then make movement decisions according to the rules of agent-1.

- Partial Predator Information Setting: The agent makes informed movement decisions based on the location of the prey, trying to catch it while making probabilistic partial informed decisions away from the predator's possible location. If the agent is certain of the predator's location, he will then make movement decisions according to the rules of agent-1.
- Combined Partial Information Setting: The agent makes uninformed movement decisions initially till he finds the location of either the prey or predator using his survey drone ability and starts to make probabilistic partial informed movement decisions conforming to agent-3 and agent-5 based on the belief states of the prey and predator.

- The prey always takes the second turn in the game regardless of the information settings. Also, the movement decision of the prey is pre-determined without any changes throughout all the various information settings. 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 all the settings.

- The predator always takes the last turn in the game regardless of the information settings. The movement pattern of the predator for the first 3 settings (Base, Complete, Partial Prey Information Settings) is simply defined by the shortest distance to the agent from its current node.
    - Easily Distracted Predator – For the other settings, the movement pattern is defined as follows: The 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.

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 as "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.

There are 6 conditions specified based on which the agent makes his decisions.

- Neighbors that are closer to the Prey and farther from the Predator.
- Neighbors that are closer to the Prey and not closer to the Predator.
- Neighbors that are not farther from the Prey and farther from the Predator.
- Neighbors that are not farther from the Prey and not closer to the Predator.
- Neighbors that are farther from the Predator.
- Neighbors that are not closer to the Predator.

- When no neighboring node leads to any of these above-stated conditions, then the agent will sit still and pray.

AGENTS:

The agent never spawns in a pre-occupied position (either on prey or predator). There are a total of 8 agents, with agent 9 being a hypothetical write-up, each with different settings and upgrades. Every agent can move along its node neighbors or stay in the same place depending on the situation. Except for agent 1 and agent 2, every other agent uses a probabilistic model to assess the position of the prey or predator or both. Every agent always knows how the prey or predator moves, irrespective of the information setting.

The entire run of the game is inside while_loop. The while_loop only terminates if any of the defined end states occur.

For partial information settings, the agent keeps track of its other entities based on an array of belief states which correspond to the probability of a certain node containing an entity. To implement this belief state, the data structure we used is a dictionary belief_states = {} where the key represents the node and values represent the probability. This dictionary is updated twice per loop with two unique equations each time. The sum of belief states at every iteration always results in 1.

The first update/equation triggers when the agent surveys a random node along with his current occupying node and updates the belief states accordingly.

$$P(PreyinM, FailedSurveyPreyatN)/P(FailedSurveyPreyatN)$$
$$= P(PreyinM)*P(FailedSurveyPreyatN|PreyinM)/P(FailedSurveyPreyatN)$$
$$P(FailedSurveyPreyatN) = 1 - P(PreyFoundN)$$
$$= P(PreyinM, FailedSurveyPreyatN)/1 - P(PreyFoundN)$$

All the belief states are calculated with the above equation if a survey at the highest probable location fails to return any value. This above equation is for updating all belief states after the agent surveys a location.

The second update happens when the agent takes the desired step in any direction based on the current belief states. This update scales the probability to the adjacent neighboring nodes of either the prey or predator based on the information setting. The iterations for this equation go as follows:

1. The agent finds either the prey or predator by surveying a random node in the environment.
2. If the survey fails, the belief states of all other nodes change according to this equation.
3. If the survey succeeds in finding either the prey or predator, then the belief state of the survey node location will be updated as 1, and rest all become 0.
4. Immediately in the next iteration, this probability of previously found node 1 will be spread/scaled out among its neighbors based on how the prey or predator moves in the belief states using this formula.
5. The agent then stops surveying randomly and surveys a node with now the highest probability of finding either the prey or predator.
6. If it finds the prey or predator again, then step 3 repeats. If the survey fails, then the probabilities change based on the 2nd equation. (prob of one node/If not prob of non-node)

$$P(PreyinNnext) = P(PreyinNnow) * P(PreyinNnext|PreyinNnow) + P(PreyinN1now) * P(PreyinNnext|PreyinN1now) + ...$$

The equation to evaluate the probabilities of prey or predator in the upcoming steps/nodes (Prey/Predator in next Node N). The above equation is calculated for all possible neighbor locations of prey or predator depending on the information setting. This equation is for assessing where the prey or predator might move to after the agent loses track of it. The previous belief states are multiplied by the actual prey or predator step probability value which scales the probability appropriately to all the neighbor nodes.

Implementation of Equations:

```
uncalculated_nodes = []
head_nodes = []
highest_prob_val_node_list = []
head_nodes = list(dict.fromkeys(head_nodes))

for i in head_nodes:
    for j in main_graph[i]:
        uncalculated_nodes.append(j)
    uncalculated_nodes.append(i)

    for k in uncalculated_nodes:
        if len(main_graph[i]) == 3:
            if current_agent_pos in uncalculated_nodes:
                temp_dict[k].append((belief_states[i] * (1 / 3)))
            elif current_agent_pos not in uncalculated_nodes:
                temp_dict[k].append((belief_states[i] * prey_move_prob))

        else:
            if current_agent_pos in uncalculated_nodes:
                temp_dict[k].append((belief_states[i] * (1 / 2)))
            elif current_agent_pos not in uncalculated_nodes:
                temp_dict[k].append((belief_states[i] * (1 / 3)))

    uncalculated_nodes.clear()

for lk in temp_dict.keys():
    if len(temp_dict[lk]) == 1:
        belief_states[lk] = temp_dict[lk][-1]
    else:
        belief_states[lk] = sum(temp_dict[lk])

belief_states[current_agent_pos] = 0
uncalculated_nodes.clear()

for i in head_nodes:
    for j in main_graph[i]:
        uncalculated_nodes.append(j)
    uncalculated_nodes.append(i)

if current_agent_pos in uncalculated_nodes:
    [uncalculated_nodes.remove(r) for r in uncalculated_nodes if r ==
current_agent_pos]

temp_dict.clear()
```

The probabilities are scaled by taking and multiplying with the previously found belief state probability of that particular node. This way, the accurate execution of belief states which produces the sum = 1 every time is produced.

```
[uncalculated_nodes.remove(r) for r in uncalculated_nodes if r ==
agent_survey_node_loc]

for j in uncalculated_nodes:
    temp_dict_2[j] = belief_states[j] / (1 -
belief_states[agent_survey_node_loc])

head_nodes.clear()
for add in uncalculated_nodes:
    head_nodes.append(add)
uncalculated_nodes.clear()

for lk in temp_dict_2.keys():
    belief_states[lk] = temp_dict_2[lk]

belief_states[agent_survey_node_loc] = 0

temp_dict_2.clear()
```

Agents Description:

Our code has two unique lists called “Uncalculated Nodes” and “Head Nodes.” The equation for assessing where the prey or predator might be in the next step if the survey fails to find either of them is calculated using these two lists.

If the survey on a particular node returns nothing, then the belief state of that node will be initialized to ‘0’, and the other probabilities of nodes with a possibility of finding prey or predator are updated according to equation 2.

Agent 1:

This agent belongs to the complete information setting of the environment. This agent uses this comprehensive information to avoid the predator and catch the prey. Agent 1 always knows where the prey and predator are, so there are no belief states to keep track of.

Agent 2:

Agent 2 works in accordance with agent 1, except the agent and predator calculate the traversal cost using ‘euclidean distance’ instead of traditional UCS. Peculiarly, calculating the distance using Euclidean always gives the path along the circular edges without heeding centered randomly connected nodes. Calculating distances this way creates an overhead offset which benefits the agent in finding the prey (regardless of the agent using Euclidean or UCS for path traversal) quickly, additionally slowing the predator’s movement. The agent here can also use traditional UCS to catch the prey to eliminate the drama of chase, but the predator movement is strictly restricted to Euclidean distance calculation (circular distance ignoring centered connections).

Agent 3:

The Agent always knows where the Predator is but does not necessarily know where the Prey is. Every time the Agent moves, the Agent can first choose a node to survey (anywhere in the graph) to determine whether or not the Prey is there. Additionally, the agent gains information about

where the Prey isn't every time it enters a node and the Prey isn't there. In this setting, the Agent needs to track a belief state for where the Prey is, a collection of probabilities for each node that the Prey is there. Every time the Agent learns something about the Prey, these probabilities need to be updated. Every time the Prey is known to move, these probabilities must be updated.

```
head_nodes = list(dict.fromkeys(head_nodes))

    for i in head_nodes:
        for j in main_graph[i]:
            uncalculated_nodes.append(j)
        uncalculated_nodes.append(i)

        for k in uncalculated_nodes:
            if len(main_graph[i]) == 3:
                if current_agent_pos in uncalculated_nodes:
                    temp_dict[k].append((belief_states[i] * (1 / 3)))
                elif current_agent_pos not in uncalculated_nodes:
                    temp_dict[k].append((belief_states[i] *
prey_move_prob))

            else:
                if current_agent_pos in uncalculated_nodes:
                    temp_dict[k].append((belief_states[i] * (1 / 2)))
                elif current_agent_pos not in uncalculated_nodes:
                    temp_dict[k].append((belief_states[i] * (1 / 3)))

        uncalculated_nodes.clear()

    for lk in temp_dict.keys():
        if len(temp_dict[lk]) == 1:
            belief_states[lk] = temp_dict[lk][-1]
        else:
            belief_states[lk] = sum(temp_dict[lk])

    belief_states[current_agent_pos] = 0
    uncalculated_nodes.clear()

    for i in head_nodes:
        for j in main_graph[i]:
            uncalculated_nodes.append(j)
        uncalculated_nodes.append(i)

    if current_agent_pos in uncalculated_nodes:
        [uncalculated_nodes.remove(r) for r in uncalculated_nodes if r ==
current_agent_pos]

    temp_dict.clear()

    highest_prob_val_node_list.clear()
    for p2 in sorted(belief_states, key=lambda k: belief_states[k],
reverse=True):
        highest_prob_val_node_list.append(p2)
    m = belief_states[highest_prob_val_node_list[0]]

    l2 = [k for k in highest_prob_val_node_list if belief_states[k] == m]

    highest_prob_val_node = random.choice(l2)
    # highest_prob_val_node = highest_prob_val_node_list[0]
    agent_survey_node_loc = highest_prob_val_node
```

```
elif 1 not in belief_states.values():

    er_check = list(belief_states.keys())
    er_check.remove(current_agent_pos)
    agent_survey_node_loc = random.choice(er_check)

if agent_survey_node_loc != current_prey_pos:
    if t > 1:

        if m == 0:
            reset()
            continue

        [uncalculated_nodes.remove(r) for r in uncalculated_nodes if r ==
agent_survey_node_loc]

        for j in uncalculated_nodes:
            temp_dict_2[j] = belief_states[j] / (1 -
belief_states[agent_survey_node_loc])

        head_nodes.clear()
        for add in uncalculated_nodes:
            head_nodes.append(add)
        uncalculated_nodes.clear()

        for lk in temp_dict_2.keys():
            belief_states[lk] = temp_dict_2[lk]

        belief_states[agent_survey_node_loc] = 0

        temp_dict_2.clear()

    # This condition below will never run after finding prey atleast once
    if t == 1:
        t = 0
        for i in belief_states.keys():
            if i == agent_survey_node_loc or i == current_agent_pos:
                belief_states[i] = 0
                continue
            belief_states[i] = (1 / 48)

elif agent_survey_node_loc == current_prey_pos:
    prey_initial_found_loc = agent_survey_node_loc
    head_nodes.clear()
    uncalculated_nodes.clear()
    for i in belief_states.keys():
        if i == agent_survey_node_loc:
            belief_states[i] = 1
            continue
        belief_states[i] = 0

sorted_pred_vals = []
sorted_prey_vals = []

total_cost_to_predator_path = BFS(current_agent_pos, current_predator_pos)
for k2 in sorted(total_cost_to_predator_path, key=lambda k:
len(total_cost_to_predator_path[k]), reverse=True):
    sorted_pred_vals.append(k2)

if prey_initial_found_loc != 0:
    highest_prob_val_node_list.clear()
```

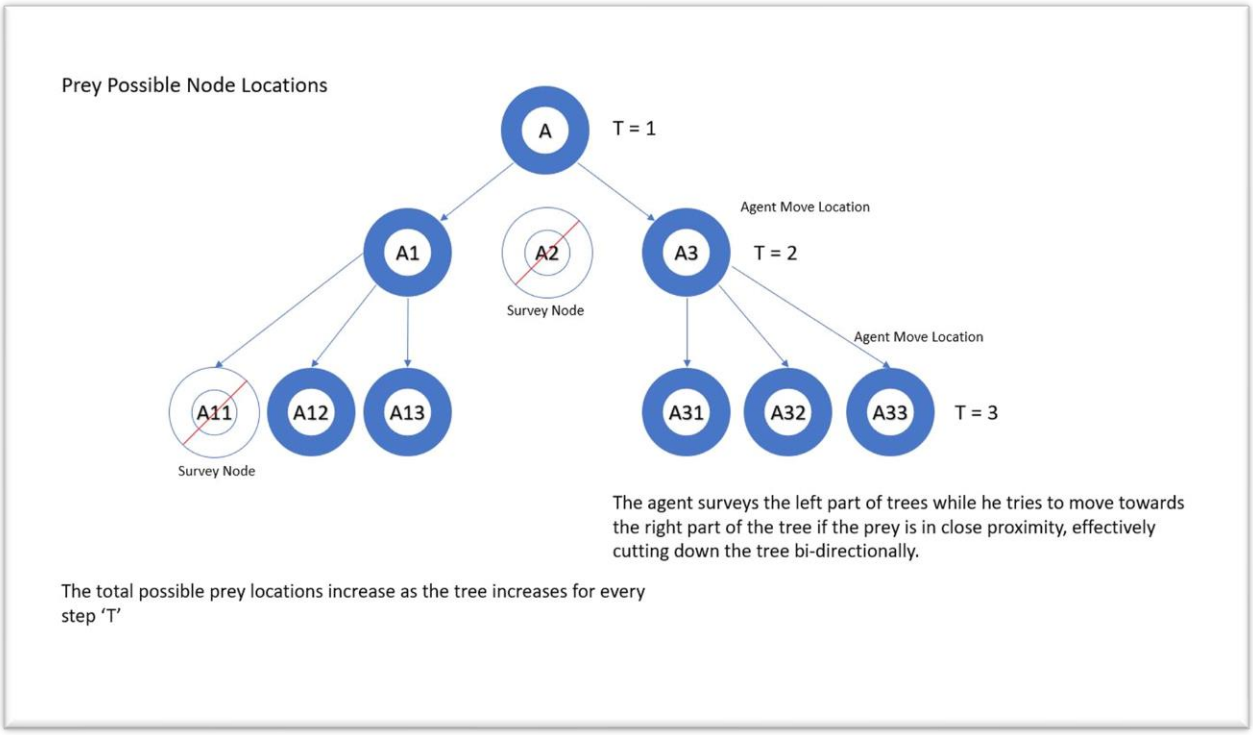
```
    for p3 in sorted(belief_states, key=lambda k: belief_states[k],
reverse=True):
        highest_prob_val_node_list.append(p3)
    highest_prob_val_node = highest_prob_val_node_list[0]

    total_cost_to_survey_prey_path = BFS(current_agent_pos,
highest_prob_val_node)

    for k1 in sorted(total_cost_to_survey_prey_path, key=lambda k:
len(total_cost_to_survey_prey_path[k]),
                     reverse=False):
        sorted_prey_vals.append(k1)
```

Agent 4:

The prey movement pattern is altered for this agent setting. The prey in this setting moves based on a different probability setting. The prey chooses to move among its neighbors with a probability of 0.3 for each neighbor and a probability of 0.1 to stay in the same place. Altering the probability of the prey this way leads to predicting the next possible nodes the prey might be with efficiency (improved accuracy) as opposed to the prey choosing positions to jump uniformly (0.25 equal probability distribution). Furthermore, this agent always tries to move to the different side of the prey's possible locations tree, which are created with equally distributed probability as opposed to the agent always going towards the node with the current highest probability of prey in belief states. (In every setting, the agent surveys the node with the highest probability and also moves towards the node with the highest probability, this functionality has been altered for this agent effectively cutting down the prey possible location tree bi-directionally).



The above diagram explains how agent 4 takes each step effectively, being better than base agent 3 in the write-up.

The remaining settings of this agent are by agent 3, except this agent is well equipped with the new probabilities of prey and is efficient in closing in on the target because of better surveying. The prey in this setting has the ability to avoid a particular node if it spots the agent there.

Agent 5:

Whenever it is this Agent's turn to move, if it is not currently specific where the Predator is, it will survey the node with the highest probability of containing the Predator and update the probabilities based on the result. After this, it will assume the Predator is located at the node with the now highest probability of containing the Predator (breaking ties first based on proximity to the Agent, then at random) and will act in accordance with the rules for Agent 1.

```
head_nodes = list(dict.fromkeys(head_nodes))

    for i in head_nodes:
        sorted_pred_to_agent_vals = []
        shortest_to_agent_frm_predator = BFS(i, current_agent_pos)
        print('shortest_to_agent_frm_predator = ',
shortest_to_agent_frm_predator)
        for s1 in sorted(shortest_to_agent_frm_predator, key=lambda k:
len(shortest_to_agent_frm_predator[k]), reverse=False):
            sorted_pred_to_agent_vals.append(s1)

        best_path_to_agent = sorted_pred_to_agent_vals[0]

        print('sorted_pred_to_agent_vals = ', sorted_pred_to_agent_vals)

        for j in main_graph[i]:
            uncalculated_nodes.append(j)
        # uncalculated_nodes.append(i)
        for k in uncalculated_nodes:
            if len(main_graph[i]) == 3:
                if current_agent_pos in uncalculated_nodes:
                    temp_dict[k].append((belief_states[i] * (1 / 2)))
                elif current_agent_pos not in uncalculated_nodes:
                    if k == best_path_to_agent:
                        temp_dict[k].append((belief_states[i] *
prob_move_to_agent))
                        continue
                    temp_dict[k].append((belief_states[i] *
prob_move_random))
            else:
                if current_agent_pos in uncalculated_nodes:
                    temp_dict[k].append((belief_states[i] * 1))
                elif current_agent_pos not in uncalculated_nodes:
                    if k == best_path_to_agent:
                        temp_dict[k].append((belief_states[i] *
prob_move_to_agent))
                        continue
                    temp_dict[k].append((belief_states[i] *
(prob_move_random * 2)))

        uncalculated_nodes.clear()
        belief_states[i] = 0

    for lk in temp_dict.keys():
        if len(temp_dict[lk]) == 1:
            belief_states[lk] = temp_dict[lk][-1]
        else:
```

```
            belief_states[lk] = sum(temp_dict[lk])

    belief_states[current_agent_pos] = 0
    uncalculated_nodes.clear()

    for i in head_nodes:
        for j in main_graph[i]:
            uncalculated_nodes.append(j)
        # uncalculated_nodes.append(i)

    if current_agent_pos in uncalculated_nodes:
        [uncalculated_nodes.remove(r) for r in uncalculated_nodes if r ==
current_agent_pos]

    temp_dict.clear()
    print('Head Nodes = ', head_nodes)
    print('Uncalculated Nodes 1 = ', uncalculated_nodes)
    print('belief_states 32 =', belief_states)
    print('belief_stats_SUM_32 = ', sum(belief_states.values()))

    highest_prob_val_node_list.clear()
    for p2 in sorted(belief_states, key=lambda k: belief_states[k],
reverse=True):
        highest_prob_val_node_list.append(p2)
    max_prob = belief_states[highest_prob_val_node_list[0]]
    print('max_prob = ', max_prob)
    # l2 = [k for k in highest_prob_val_node_list if belief_states[k] == m]
    # print('l2 = ', l2)
    # highest_prob_val_node = random.choice(l2)
    highest_prob_val_node = highest_prob_val_node_list[0]
    agent_survey_node_loc = highest_prob_val_node

elif 1 not in belief_states.values():
    print('1 I came here')
    er_check = list(belief_states.keys())
    er_check.remove(current_agent_pos)
    agent_survey_node_loc = random.choice(er_check)

print('agent_survey_node_loc = ', agent_survey_node_loc)

if agent_survey_node_loc != current_predator_pos:
    if t > 1:
        print('Uncalculated Nodes 2 = ', uncalculated_nodes)
        print('agent_survey_node_loc 2 = ', agent_survey_node_loc)

        if max_prob == 0:
            reset()
            continue

        [uncalculated_nodes.remove(r) for r in uncalculated_nodes if r ==
agent_survey_node_loc]

        for j in uncalculated_nodes:
            temp_dict_2[j] = belief_states[j] / (1 -
belief_states[agent_survey_node_loc])

        head_nodes.clear()
        for add in uncalculated_nodes:
            head_nodes.append(add)
        uncalculated_nodes.clear()
```

```
        for lk in temp_dict_2.keys():
            belief_states[lk] = temp_dict_2[lk]

        belief_states[agent_survey_node_loc] = 0

        temp_dict_2.clear()

        print('belief_states 44 =', belief_states)
        print('belief_stats_SUM_44 = ', sum(belief_states.values()))

    # This condition below will never run after finding prey atleast once
    if t == 1:
        t = 0
        for i in belief_states.keys():
            if i == agent_survey_node_loc or i == current_agent_pos:
                belief_states[i] = 0
                continue
            belief_states[i] = (1 / 48)
        # if n < 1:
        #     n = n + 1
        # print('n = ', n)

elif agent_survey_node_loc == current_predator_pos:
    print('I found predator here once!')
    predator_initial_found_loc = agent_survey_node_loc
    head_nodes.clear()
    uncalculated_nodes.clear()
    for i in belief_states.keys():
        if i == agent_survey_node_loc:
            belief_states[i] = 1
            continue
        belief_states[i] = 0

sorted_prey_vals = []
sorted_pred_vals = []

# Find prey dist to agent here
total_cost_to_prey_path = BFS(current_agent_pos, current_prey_pos)
for k2 in sorted(total_cost_to_prey_path, key=lambda k:
len(total_cost_to_prey_path[k]), reverse=False):
    sorted_prey_vals.append(k2)

print('sorted_prey_vals = ', sorted_prey_vals)

if predator_initial_found_loc != 0:
    highest_prob_val_node_list.clear()

    for p3 in sorted(belief_states, key=lambda k: belief_states[k],
reverse=True):
        highest_prob_val_node_list.append(p3)
    highest_prob_val_node = highest_prob_val_node_list[0]
    total_cost_to_survey_pred_path = BFS(current_agent_pos,
highest_prob_val_node)

    for k1 in sorted(total_cost_to_survey_pred_path, key=lambda k:
len(total_cost_to_survey_pred_path[k]),
                     reverse=True):
        sorted_pred_vals.append(k1)
```

Agent 6:

The prey movement pattern is altered for this agent setting. The prey movement alteration is the same as the agent 4 setting. Additionally, the orthodox predator movement of early agent settings is also altered. This agent partially conforms to agent 5 typically as this agent knows where the prey is and also completely performs per agent 4. Still, it is harder to keep track of the pursuing predator as the predator's probabilities of choosing its neighbors are skewed arbitrarily. The prey in this setting also has the ability to avoid a particular node if it spots the agent there.

Agent 7:

Whenever it is this Agent's turn to move, if it is not currently specific where the predator is, it will survey Agent 5. If it knows where the Predator is but not the Prey, it will analyze the following Agent 3. As before, however, this Agent only surveys once per round. Once probabilities are updated based on the survey results, the Agent acts by assuming the Prey is at the node of the highest probability of containing the Prey and guessing the Predator is at the node of the highest probability of having the Predator. It then applies the actions of Agent 1.

Agent 9 Hypothetical:

This agent runs the prey and the predator 100's of times before the actual game starts and stores all the probabilities of nodes taken by the two entities in an array. When the actual game commences, the agent can then use this pre-created array of information to predict where the prey or predator might be in the next step, effectively seeing the future. (Agent moving towards the future positions of prey/predator).

Base Agent:

The base agent completely ignores the predator and greedily chases the prey.

Prey Movement:

```
# PREY MOVEMENT
if len((main_graph[current_prey_pos] + [current_prey_pos])) == 4:
    current_prey_pos_list = random.choices((main_graph[current_prey_pos] +
[current_prey_pos]),
                                           [25, 25, 25, 25], k=1)
else:
    current_prey_pos_list = random.choices((main_graph[current_prey_pos] +
[current_prey_pos]),
                                           [33.33, 33.33, 33.33], k=1)

current_prey_got_pos = current_prey_pos_list[-1]

if current_prey_got_pos == current_agent_pos:
    # Might need to change probabilities for prey jump nodes here
    pass
else:
    current_prey_pos = current_prey_got_pos
```

Predator Movement:

```
# PREDATOR MOVEMENT
def predator_movement():
    cur_pred_node_pos = current_predator_pos
    checked_nodes = []
```

```
    unchecked_nodes = [[cur_pred_node_pos]]
    heap.heapify(checked_nodes)
    while len(unchecked_nodes) != 0:
        m = unchecked_nodes.pop(0)
        last_node = m[-1]
        for n in main_graph[last_node]:
            if n in checked_nodes:
                continue
            path = list(m)
            path.append(n)
            unchecked_nodes.append(path)
            if n == current_agent_pos:
                # print('I see Agent!')
                return path
        checked_nodes.append(last_node)

predator_path = predator_movement()
# print('Predator Path = ', predator_path)
current_predator_pos = predator_path[1]
```

Grey Box Questions:

Q3: Probability updates for the predator differ from the prey in the following way. The prey tries to occupy a node with uniform probability among its current node neighbors, including self-position. In comparison, the predator occupies the node, which gets the predator closer to the agent with a 0.6 probability, while it may get distracted by other nodes with a 0.4 probability.

Q4: For the general setting, the agent moves and surveys towards or away from the node with the current highest probability of finding prey or predator according to recently updated belief states. But the tree expands as the agent needs to survey accurately. So, to effectively cut down the prey possible nodes tree, it is smart to move the agent towards one end of the tree while surveying the other end of the tree, and if he spots the prey, he will then move one step closer to where the prey is. This setting can be fine-tuned further by making the agent move towards the future probable nodes the prey might occupy from a current given state. The same goes for the predator as well.

Q5: The node to survey would be best always if the chosen node to survey is amongst the highest probable nodes containing the entity.

Defect Drone Agent Setting

Before when the survey is successful, the belief state of that node containing either the prey or predator will be set to 1, and all other nodes' belief states will be set to 0. But here, due to the fact that the survey is not 100% accurate and there is a slight 0.1 chance of a mistake, the actual value to set for that particular node if it finds an entity will be equal to 0.9 instead of 1. The equation to update the other probabilities after the survey remains the same. This way, the setting can factor in the chance of defect probability precisely.

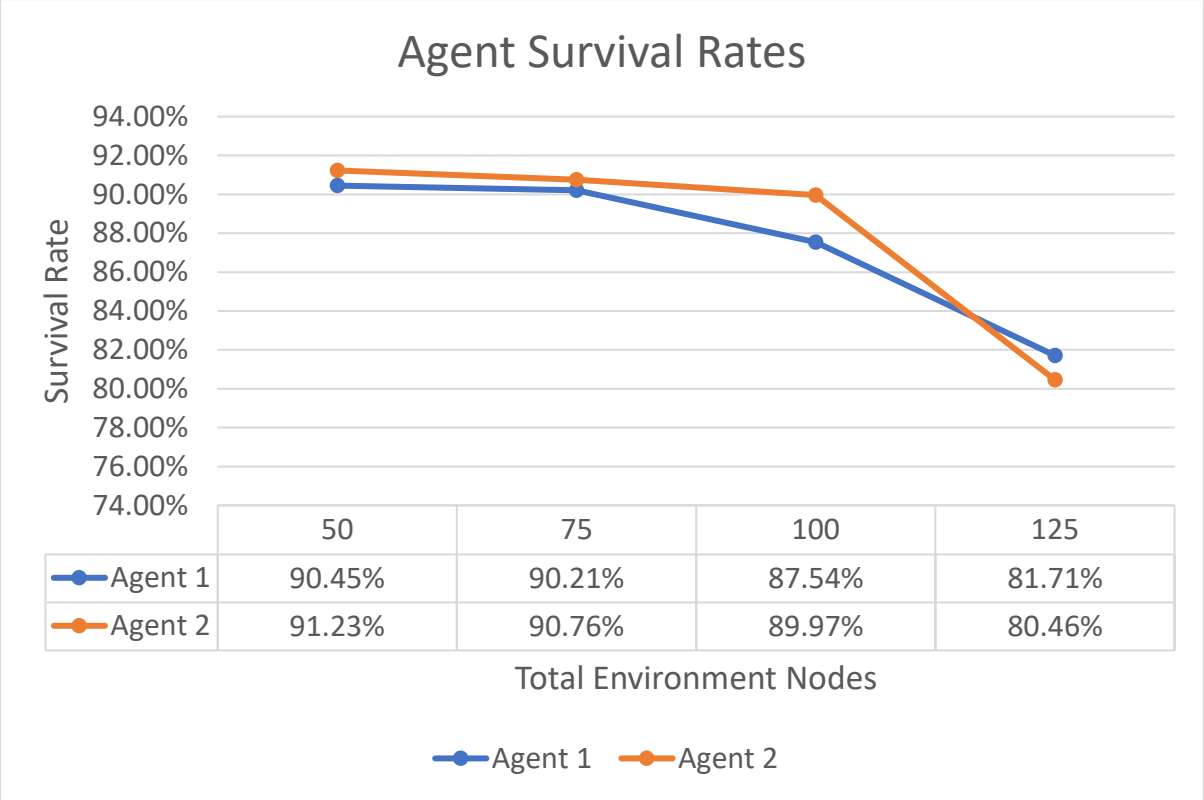
BONUS:

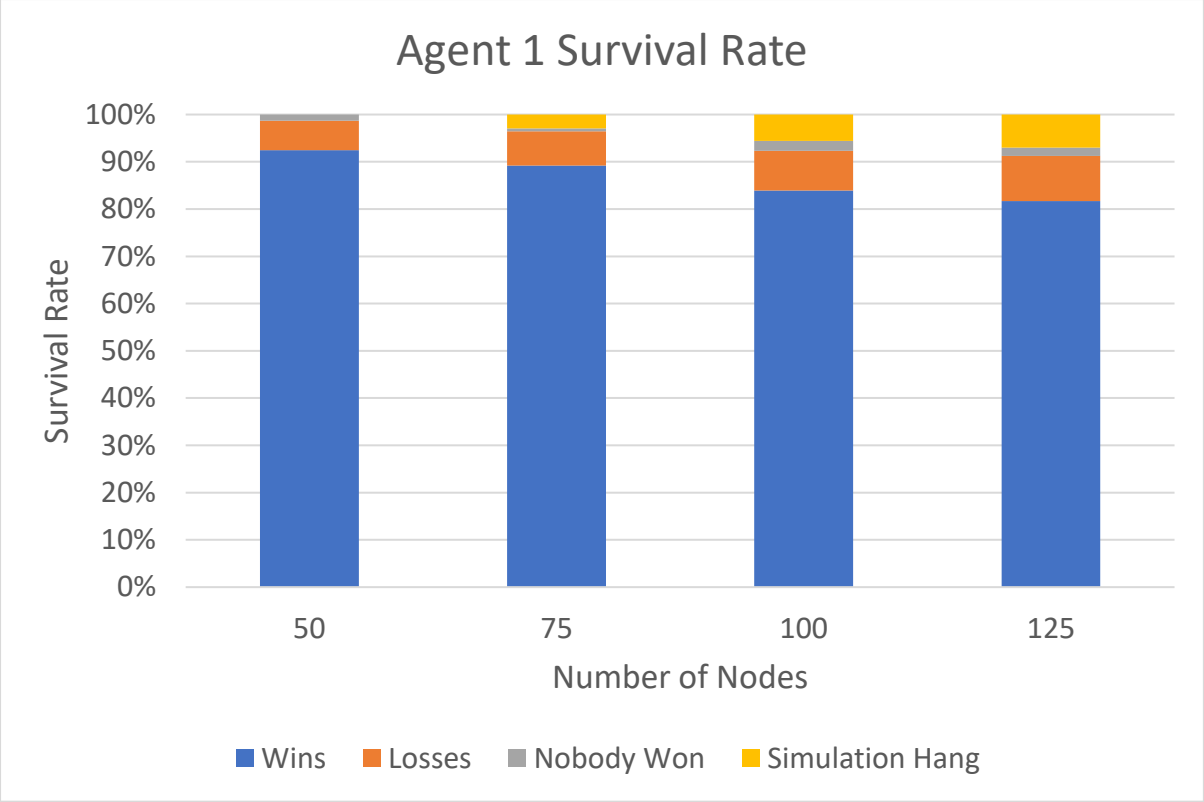
It is better to move randomly in any direction while simultaneously surveying because the agent always knows that the predator is coming straight for him. It would be unwise to stand still and keep surveying until the agent finds any entity. With this in mind, it is also possible the agent might be accidentally moving towards the predator, but the same goes for prey too. An ideal strategy would be to move a few steps initially until the survey returns successfully. If the surveying keeps taking too

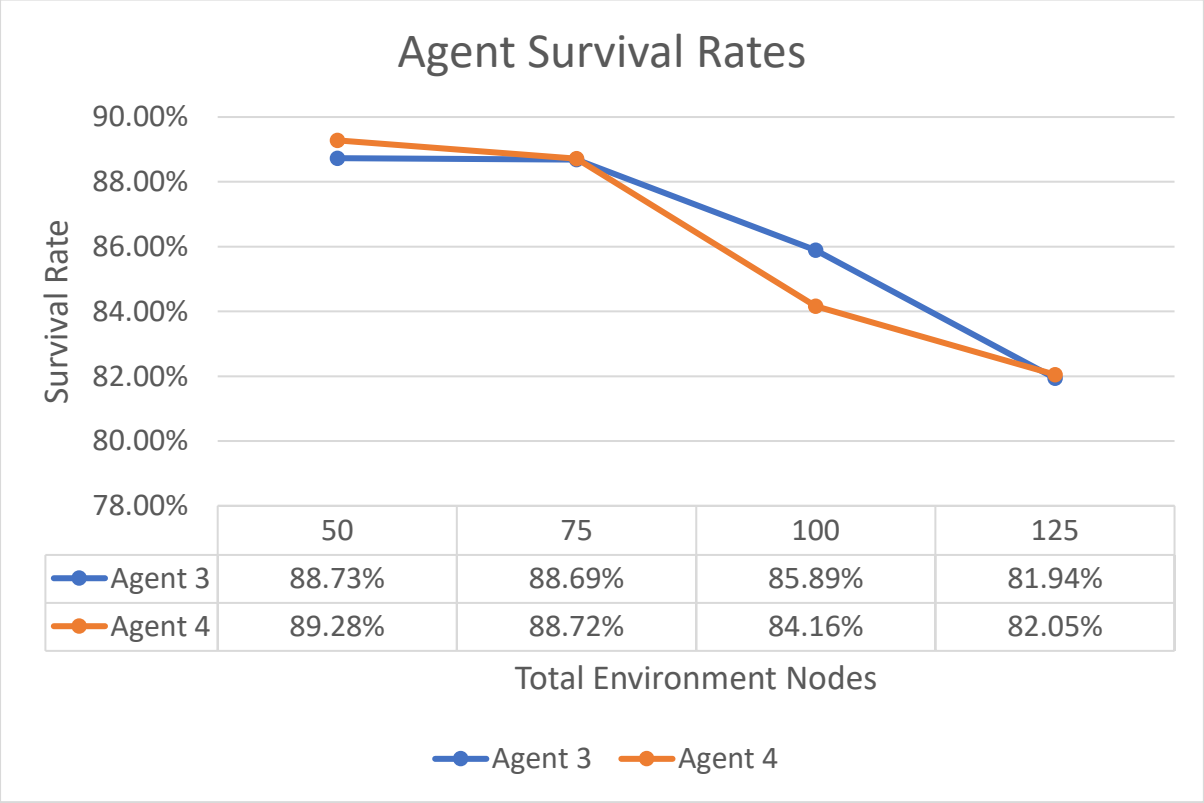
long to find any entity, then after a few initial steps, it would be wise to stop and only survey. By moving initially and still being alive infers that the predator might be a little far from the agent's position. So, the agent might get a little overhead to survey for prey after the initial wiggle steps.

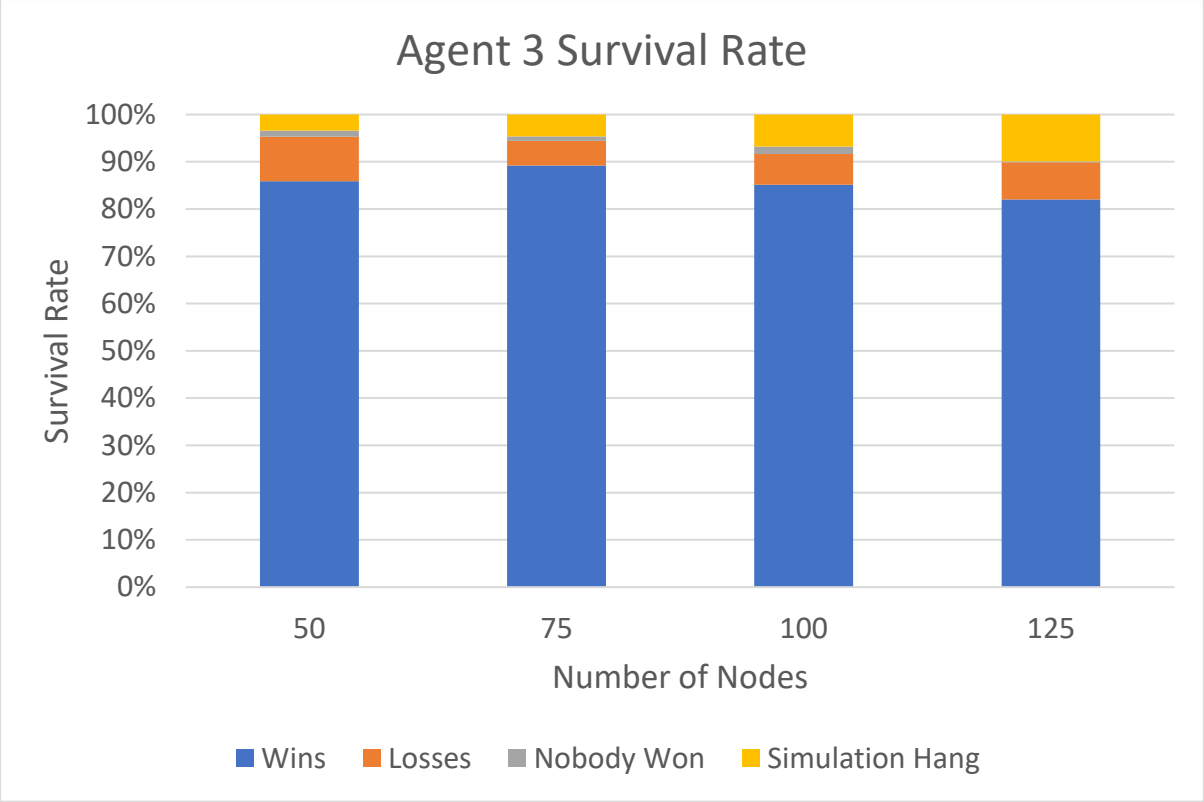
Comparisons of Agents

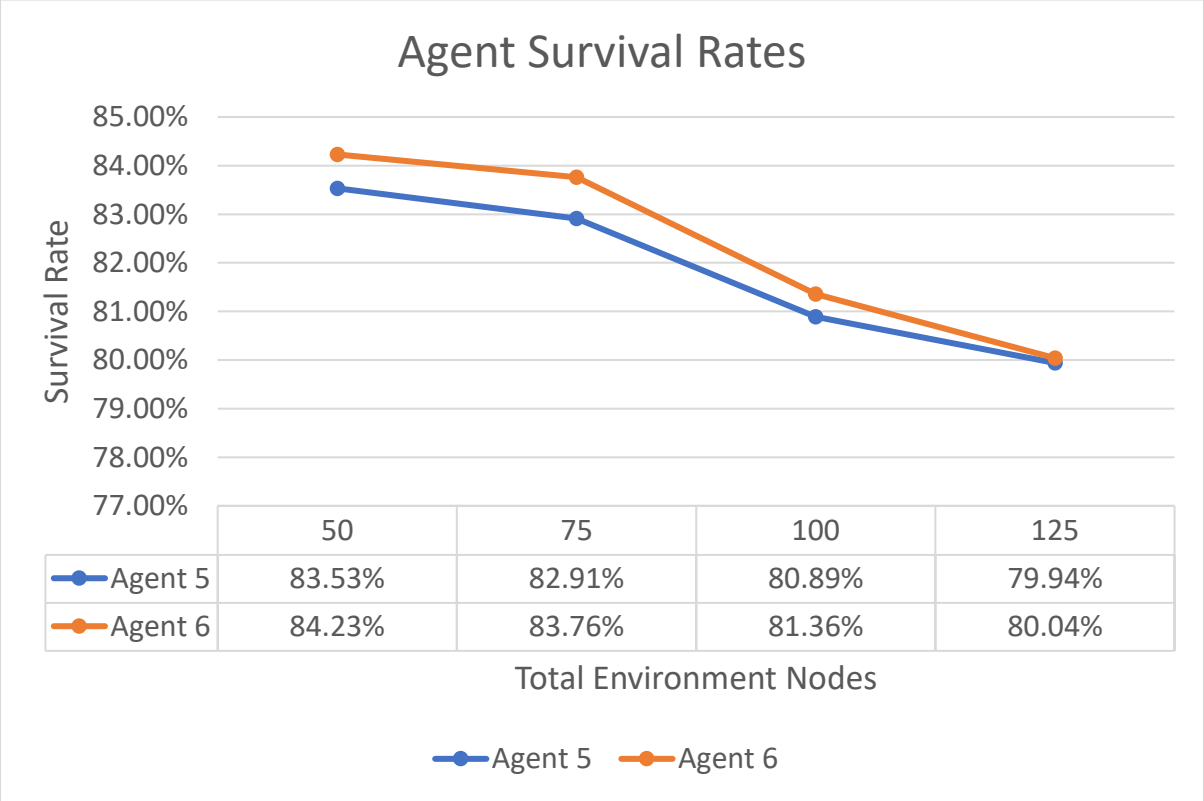
Agent 1 and Agent 2 survivability comparison chart as the number of nodes (i.e., the environment) increases.

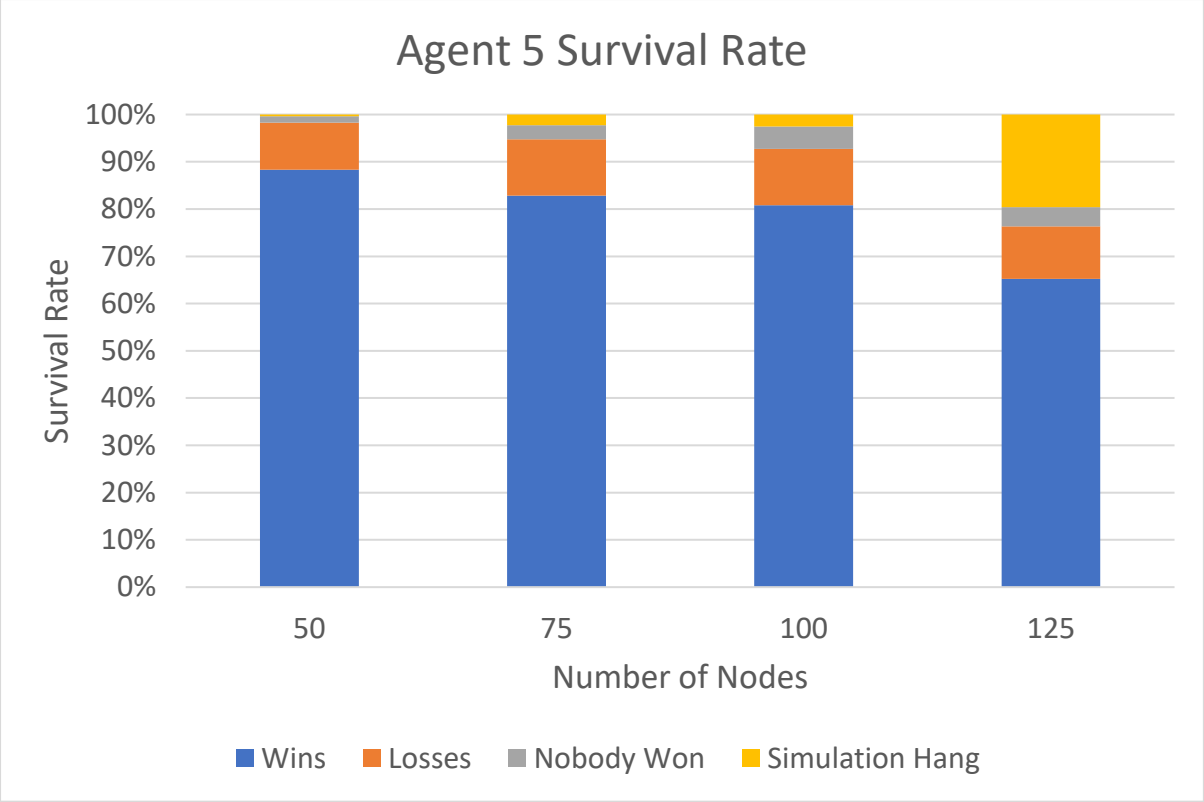


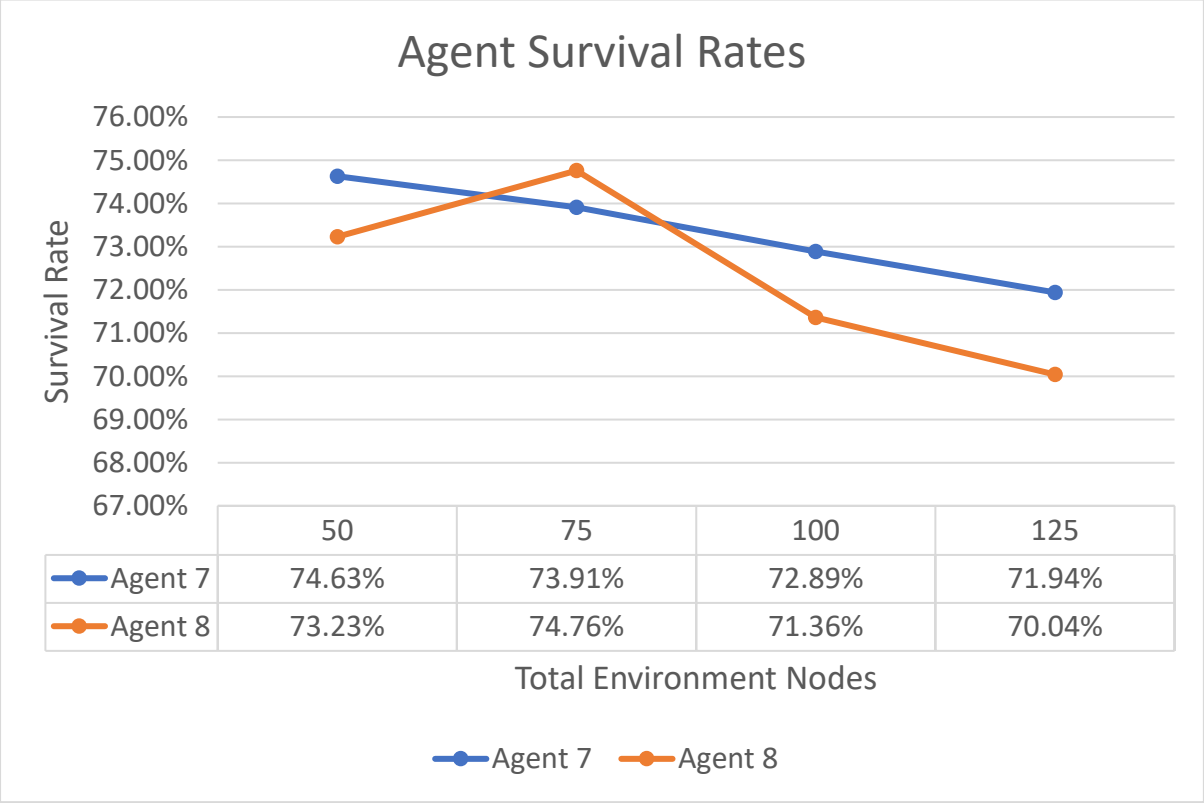

Agent 1 Survival Rate
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Survival Rate
50
75
100
125
Number of Nodes
Wins
Losses
Nobody Won
Simulation Hang

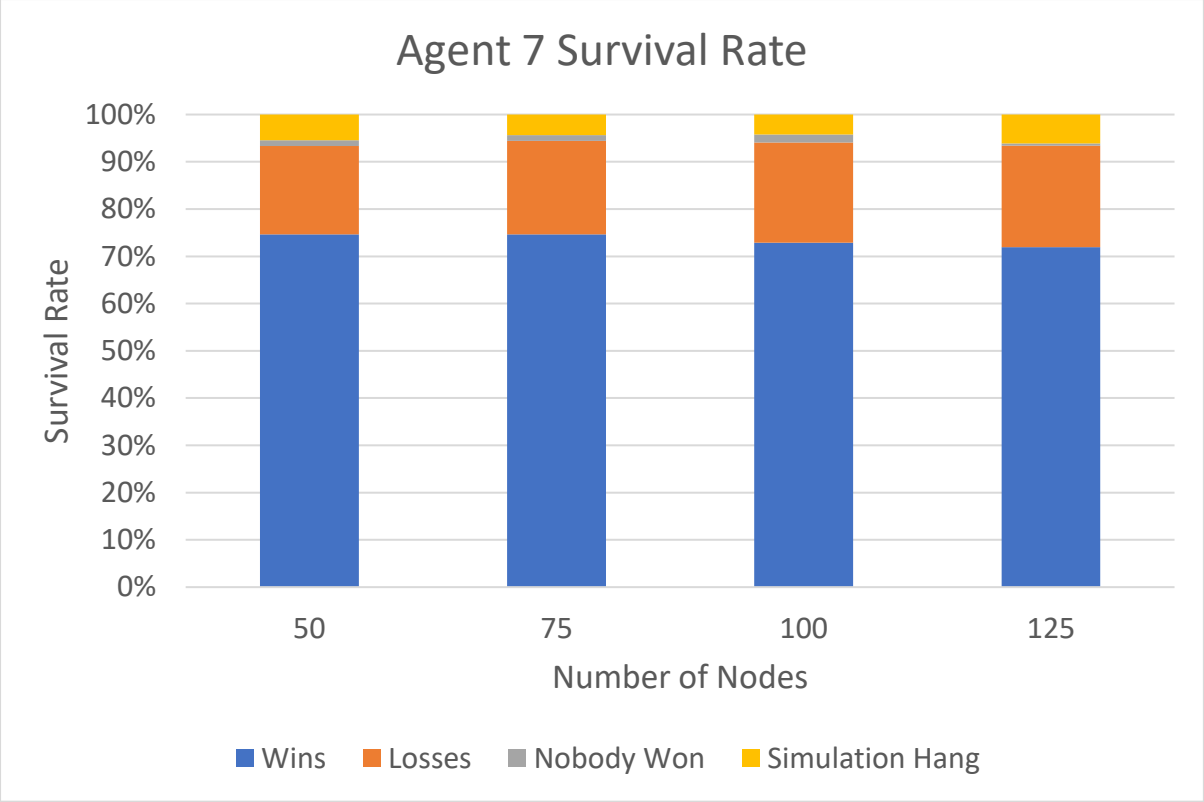

Agent Survival Rates
90.00%
88.00%
86.00%
84.00%
82.00%
80.00%
78.00%
Survival Rate
50
75
100
125
Agent 3
88.73%
88.69%
85.89%
81.94%
Agent 4
89.28%
88.72%
84.16%
82.05%
Total Environment Nodes
Agent 3
Agent 4

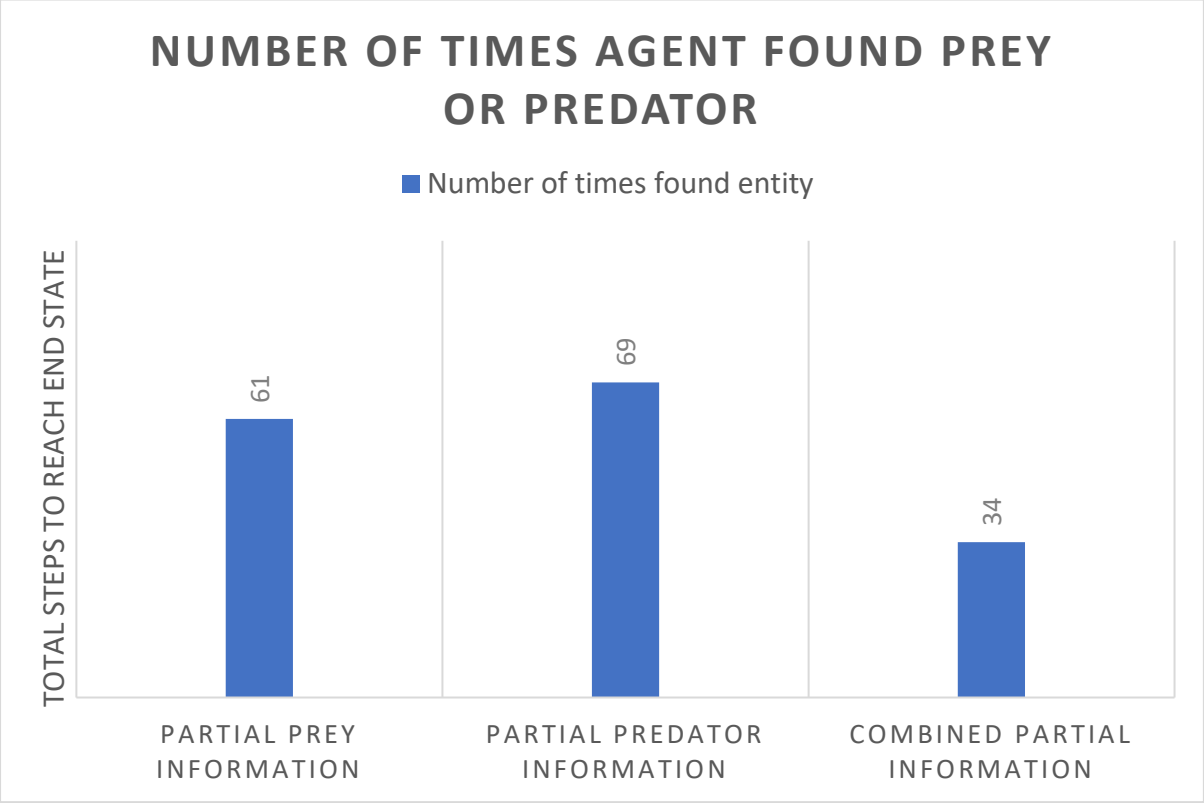

Agent 3 Survival Rate
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Survival Rate
50
75
100
125
Number of Nodes
Wins
Losses
Nobody Won
Simulation Hang


Agent Survival Rates
85.00%
84.00%
83.00%
82.00%
81.00%
80.00%
79.00%
78.00%
77.00%
Survival Rate
50
75
100
125
Agent 5
83.53%
82.91%
80.89%
79.94%
Agent 6
84.23%
83.76%
81.36%
80.04%
Total Environment Nodes
Agent 5
Agent 6

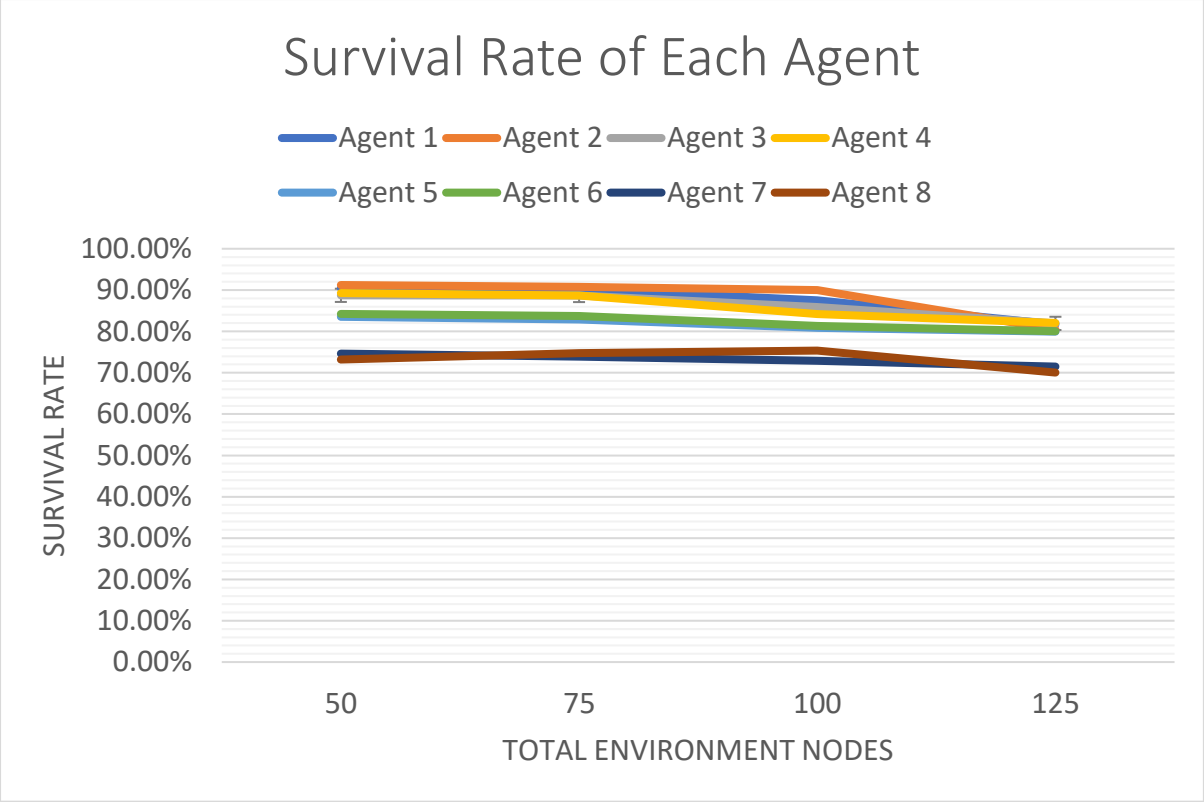

Agent 5 Survival Rate
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Survival Rate
50
75
100
125
Number of Nodes
Wins
Losses
Nobody Won
Simulation Hang


Agent Survival Rates
76.00%
75.00%
74.00%
73.00%
72.00%
71.00%
70.00%
69.00%
68.00%
67.00%
Survival Rate
50
75
100
125
Agent 7
74.63%
73.91%
72.89%
71.94%
Agent 8
73.23%
74.76%
71.36%
70.04%
Total Environment Nodes
Agent 7
Agent 8





Average num of times found the entity.



Submitted by:

Dheekshith Dev Mekala

dm1653

Sai Surya R

sr1789