520 : CIRCLE OF LIFE

A Project Report by:
- Saransh Sharma (ss4368)

LINK TO COLAB [FORMULAE MENTIONED IN CODE AS WELL]

Environment Setup

The environment consists of a graph of 50 nodes[0-49], connected via edges in a circularly linked manner. We have added shortcut-edges for a random node ‘i’ and any node with degree 2 in the range [i-5, i+5] such that there are no nodes that with a degree >3. This is repeated till no more edges can be added.

There are 3 players on the above mentioned graph environment – Predator, Agent, Prey – they can move along the edges, moving from any node to any of its neighbours in a single iteration. Goals for each of the players is as follows:

- Predator – Wants to catch the Agent
- Agent – Wants to catch the Prey
- Prey – Moves in a fixed manner without any motive

If the Predator or Agent fulfils their goal, the game ends – the Predator & Agent can’t exist in the same node, the Agent & Prey can’t exist in the same node.

The Predator and Prey can co-exist in the same node.

Their movements are in the following order –

1. Agent
2. Prey
3. Predator

Design Choices

I have taken an Object Oriented approach using Python (v3.7) for our implementation of the given task, this OOP approach helps us in easy refactoring of the code if required and also makes adding attributes, properties and functions easier down the line. The code is also highly functional and as such consists of a number of functions which improve ease of implementation and readability.

There are 4 major classes that have been implemented:

- **Graph** Class – Consists of the skeleton of the environment (the graph) which has been implemented using an Adjacency List (reduces time complexity while performing various functions) and consists of various helper functions like *get_deg(), get_next_moves(), BFS()* (the search algorithm that is used), *get_path(), get_valid_neighbours()* etc.

- **Prey** Class – Consists of the movement logic for the Prey (moves to one of its neighbours uniformly at random).

- **Predator** Class – Consists of the movement logic for the Predator (greedily trying to minimise its distance to the Agent).

- **Agent** Classes – Consists of the logic for each of the agents that are specified and the agents that I have proposed (Agent1 → Agent8). Odd numbered agents have been implemented according to given instructions and even agents utilise different novel approaches to try and beat specified approaches.

Search Algorithm

<u>Breadth First Search</u>

- Traverses nodes at equal depth from a parent node before moving forward in depth (level by level) and in this way, at some time, the fringe will contain all nodes that are equidistant from the root and popping these nodes, at some time the fringe will contain all nodes 1 level down.
- The data structure that we use for the *Fringe* in this case is a **Queue** as we want to explore the oldest nodes first and it’s a **FIFO** (First In First Out) structure.
- It generally has a bigger memory footprint than DFS.
- If a path exists, and given BFS’ property to explore all nodes on the same level first before going deeper, it guarantees that the path it finds is the **optimal/shortest** one.

Agents

Agents are our way of navigating the graph environment using BFS along with some intelligence on top.

For each agent we define certain criteria to measure its performance – Wins, Losses, Hung Runs and Survival Rate (defined by number of steps that the agent stayed alive)

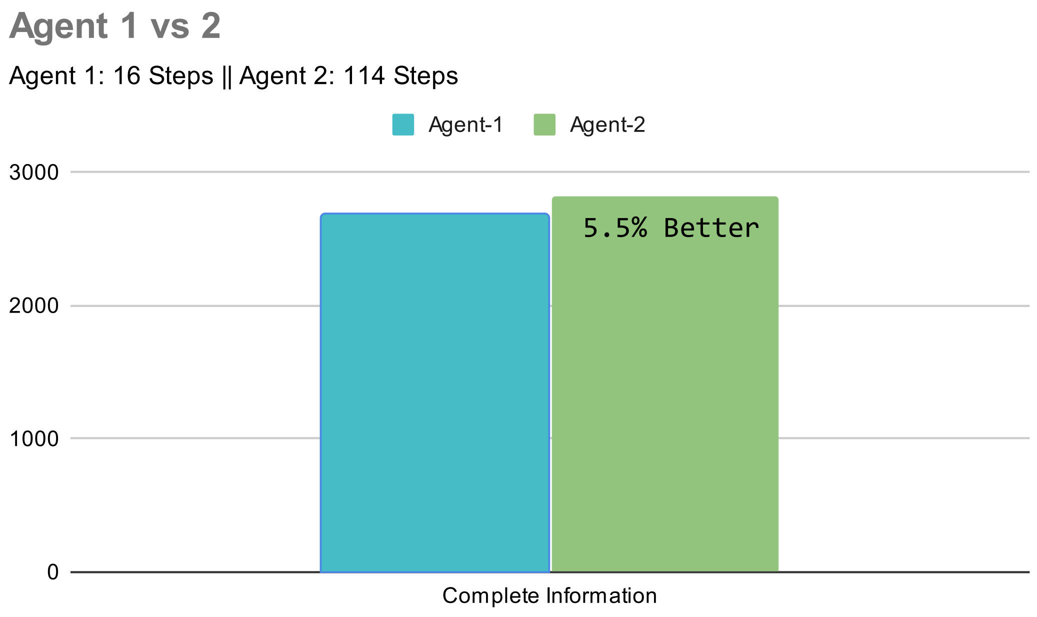
The Complete Information Setting

AGENT-1

- For Agent-1 we have complete information of where the prey & predator are always. It follows certain rules for its decision making.
- Since it has complete information & it prioritises moving closer to the prey while moving farther from the predator (if possible), it rarely gets caught.

AGENT-2

- For Agent-2 as well we have complete information of where the prey & predator are always.
- For Agent-2, I have opted to prioritise running away from the predator over running after the prey.



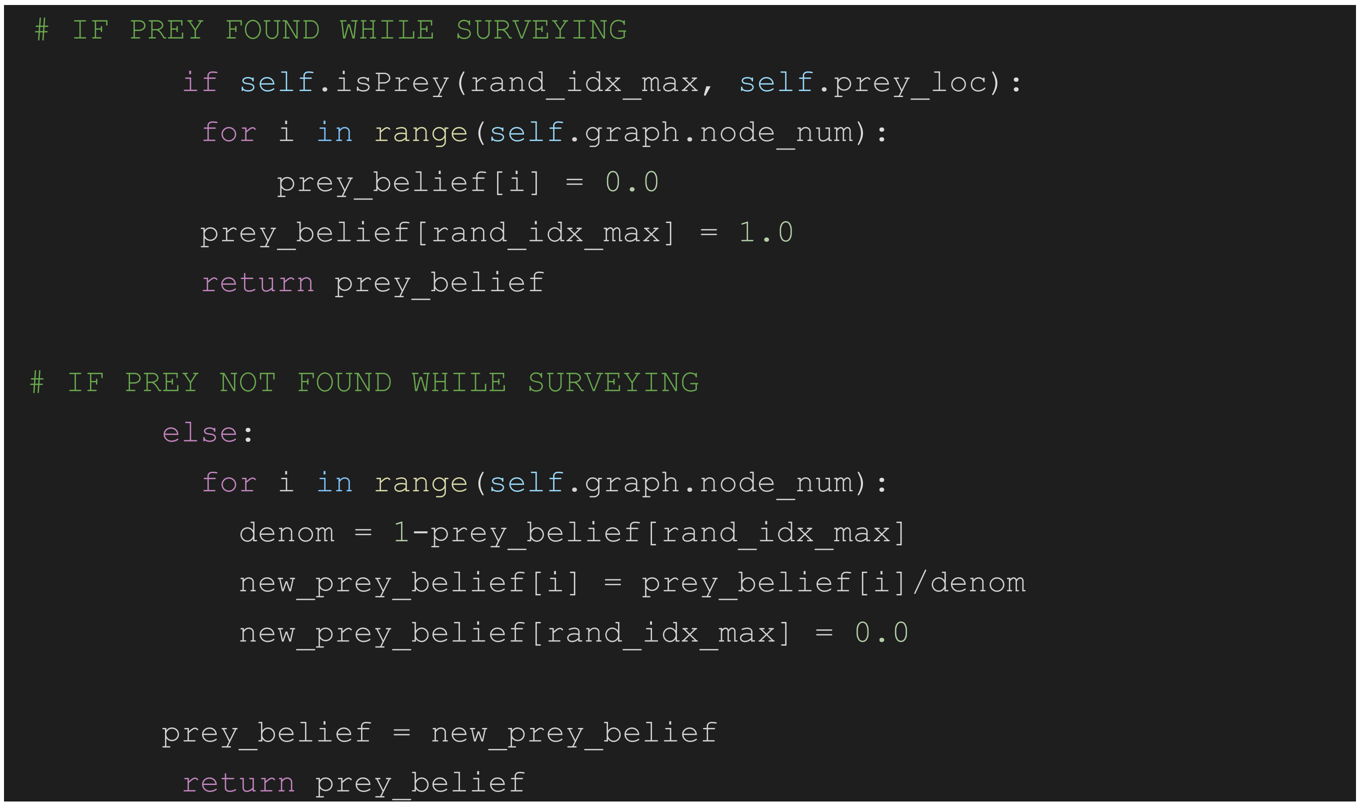
The Partial Prey Information Setting

AGENT-3

- For Agent-3, we don’t know where the Prey is but we know where the predator is at all times, but we have the ability to survey a node of our choice before the agent moves – this gives us additional information about the belief of the prey’s location.
- We initially have $Probability(Prey\ in\ Node) = 1/49$ for all nodes except the one where the agent gets initialised
- We use given formula to calculate & update the Probability of the prey being in a node at given time using the following:

  - *After Survey*

$$P(\text{Prey in Node i}) = \frac{P(\text{ Prey in } i) * P(\text{ Prey not in Surveyed Node } \mid \text{ Prey in } i)}{1 - P(\text{ Prey in Surveyed Node })}$$



```
# IF PREY FOUND WHILE SURVEYING
        if self.isPrey(rand_idx_max, self.prey_loc):
         for i in range(self.graph.node_num):
             prey_belief[i] = 0.0
         prey_belief[rand_idx_max] = 1.0
         return prey_belief

# IF PREY NOT FOUND WHILE SURVEYING
       else:
         for i in range(self.graph.node_num):
           denom = 1-prey_belief[rand_idx_max]
           new_prey_belief[i] = prey_belief[i]/denom
           new_prey_belief[rand_idx_max] = 0.0

       prey_belief = new_prey_belief
        return prey_belief
```

- *After Agent Moves*

$$P(\text{Prey in Node i}) = \frac{P(\text{Prey in } i) * P(\text{ Prey not in Current Location } \mid \text{ Prey in } i)}{1 - P(\text{Prey in Current Location})}$$

```
 # IF PREY FOUND AFTER MOVING → GAME OVER
   if self.isPrey(self.loc, prey_loc):
        for i in range(self.graph.node_num:
            prey_belief[i] = 0.0
        prey_belief[self.loc] = 1.0
        return prey_belief

# IF PREY NOT FOUND AFTER MOVING
      else:
        for i in range(self.graph.node_num):
            denom = 1.0-prey_belief[self.loc]
            new_prey_belief[i] = prey_belief[i]/denom
            new_prey_belief[self.loc] = 0.0

      prey_belief = new_prey_belief
      return prey_belief
```

- *After Prey Moves*

$$\text{For all j in } (0, 49) -> P(\text{Prey in Node } j_{next}) = \sum_{i=0}^{49} P(\text{Prey in } i) * P(\text{Prey in } j_{next} \mid \text{ Prey in } i)$$

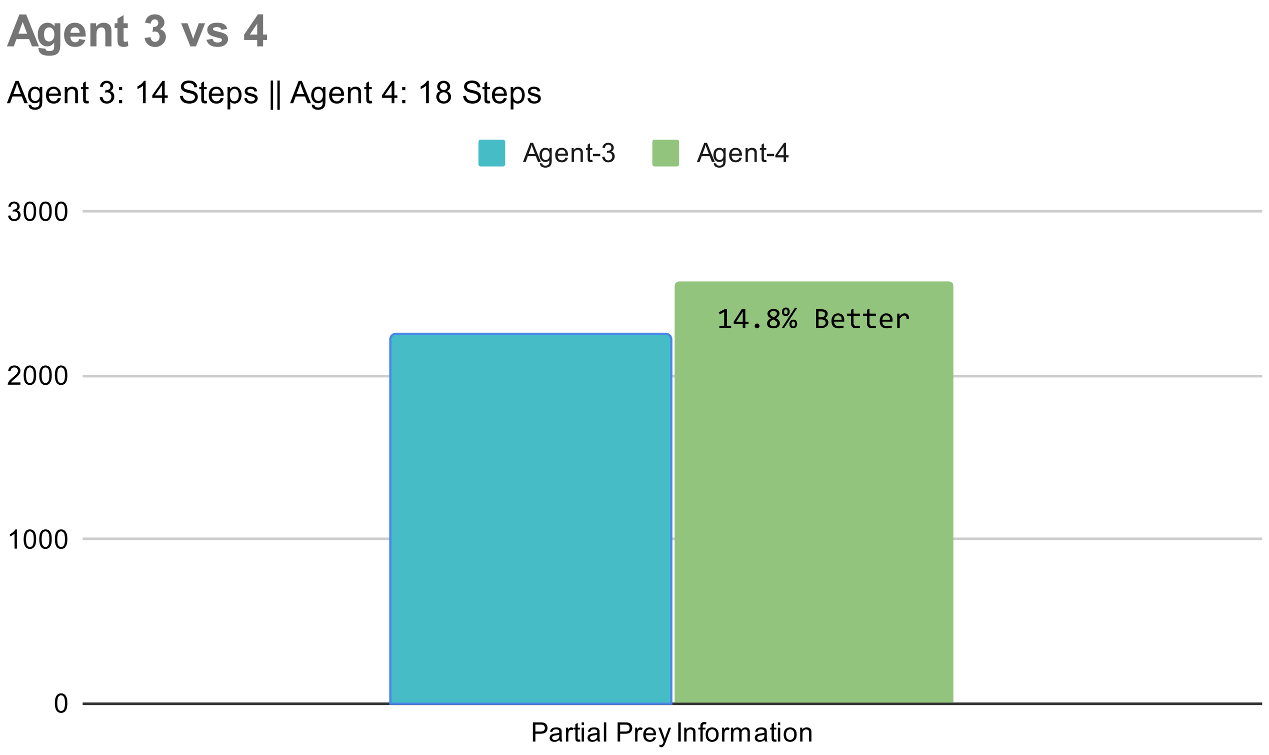
```
for i in range(self.graph.node_num):
   nbrs_i = self.graph.get_next_moves(i)
   nbrs_i.append(i)
   for nbr in nbrs_i:
      denom = self.graph.get_deg(nbr) + 1
      new_prey_belief[i] += prey_belief[nbr] / denom

   prey_belief = new_prey_belief
```

```
        prey_belief = self.agent_move_prey_belief(prey_loc, prey_beli
ef)
        return prey_belief
```

***NOTE* : FORMULAE VERIFIED WITH Dr.Cowan & our TA**

AGENT-4

- Agent-4 has the same constraints as Agent-3 in that it doesn't know where the prey is but knows where the predator is always.
- APPROACH : I find the prey belief 2 steps in the future and have my agent move accordingly. This is done by multiplying the current belief with the transition probabilities 2 times.



The Partial Predator Information Setting

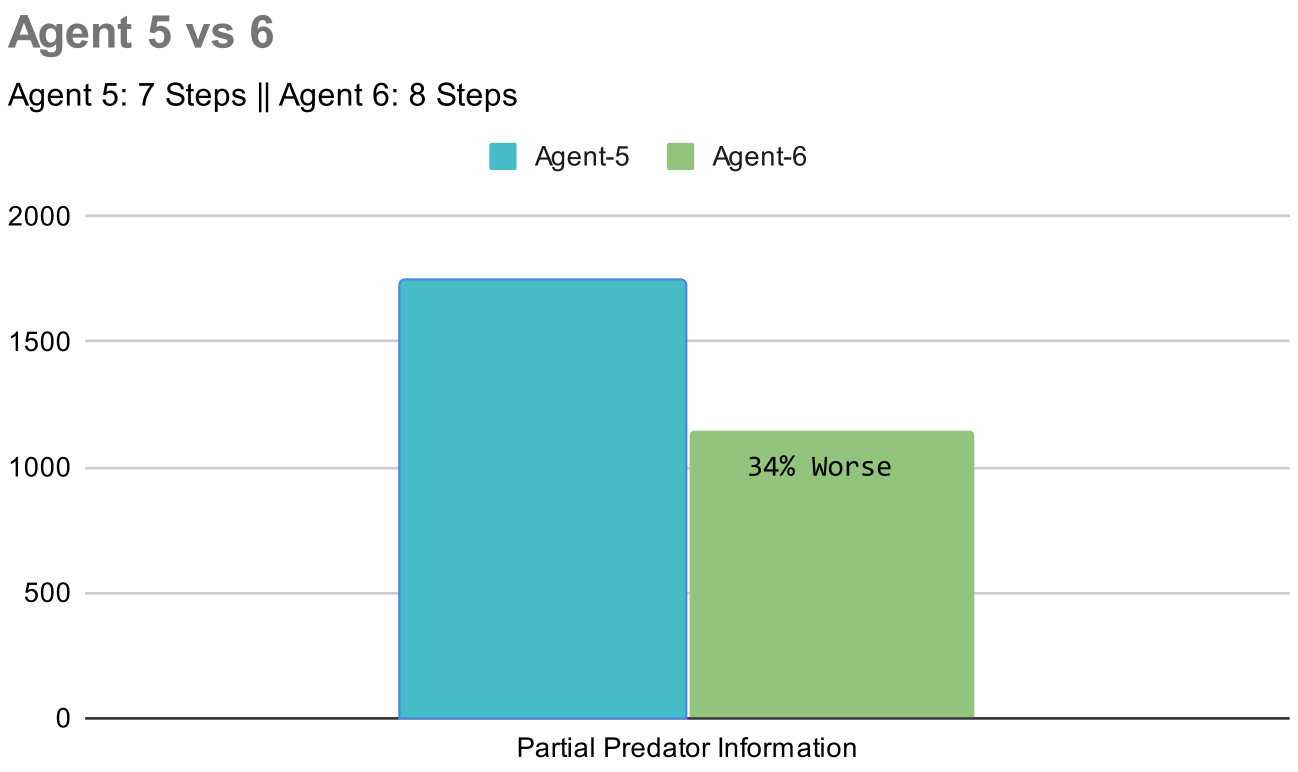
AGENT-5

- For Agent-5, we always know where the prey is but don't know where the (distracted)predator is.
- We have the ability to survey before the agent moves to gain extra information.
- We use given formula to calculate & update the Probability of the prey being in a node at given time using the following:
    - *After Survey* – The mathematical formula is very similar to surveying done for the prey. The main difference here is that the predator doesn't always have the choice to stay in its current position (it does so with certain probability)

```
if self.isPred(rand_idx_max, self.pred_loc):
            for i in range(self.graph.node_num):
                pred_belief[i] = 0.0
            pred_belief[rand_idx_max] = 1.0
            return pred_belief
        else:
            for i in range(self.graph.node_num):
                denom = 1-pred_belief[rand_idx_max]
                new_pred_belief[i] = pred_belief[i]/denom
                new_pred_belief[rand_idx_max] = 0.0
        pred_belief = new_pred_belief
        return pred_belief
```

AGENT-6

- Agent-6 has the same constraints as Agent-5 in that it doesn’t know where the predator is but knows where the prey is always.
- APPROACH : I chose to create a ‘force-field’ so to speak around the agent so that it does the following :
    - It starts off by chasing after the prey but if it senses(according to maximum predator belief) that the predator is in close proximity (within 5 nodes), it changes its priority to running away from the predator.



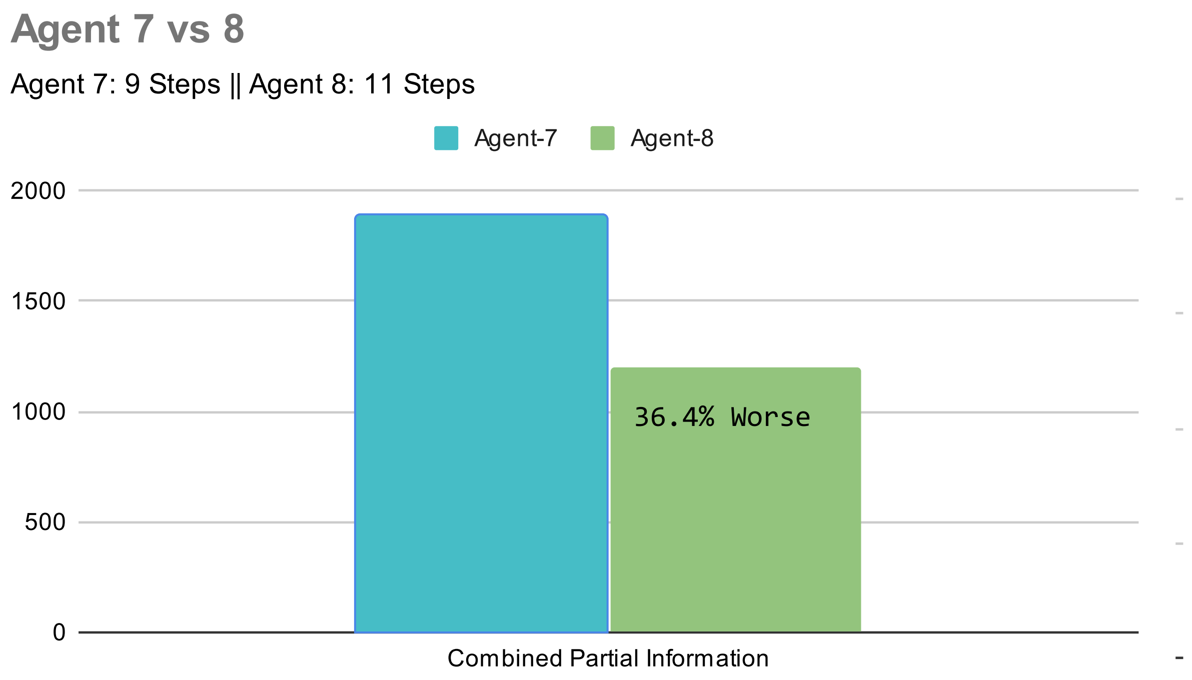
The Combined Partial Information Setting

AGENT-7

- For Agent-7, we don't know the location of the Prey and the Predator.
- We have the ability to survey (only once) and we choose to survey for the predator till we are sure of its location, at which point we can survey for the Prey.
- We keep a track of belief states for both Prey and Predator in this setting.
- The mathematical formulation for the beliefs is just a combination of the previous 2 cases to find the beliefs independently and using them together.

AGENT-8

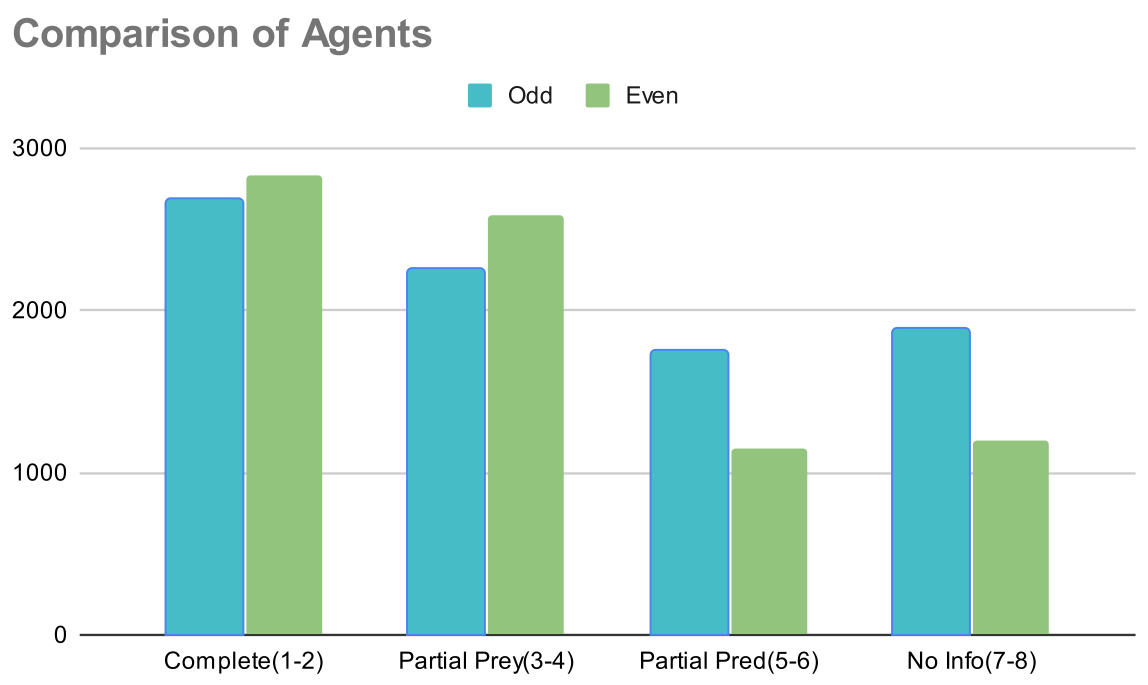
- Agent-8 has same conditions as Agent-7 wherein there is a total lack of information about the prey and predator along with the ability to survey.



ANALYSIS

- I ran 3000 simulations on different graphs for all the data that has been recorded. (Keeping threshold for hung simulations to be 5000, as mentioned by Dr.Cowan in class)

COMPARISON OF ALL AGENTS



Times that Prey was Found:

- Agent-3: 2.3 Times
- Agent-4: 4.8 Times

Times that Predator was Found:

- Agent-5: 6.7 Times
- Agent-6: 8.9 Times

Times that Prey/Predator were Found:

- Agent-7: Prey - 0.009 ; Predator - 7.8 Times
- Agent-8: Prey – 0.008 ; Predator – 6.8 Times

Observations

- We can see from the above data that making a forcefield in this case wasn’t a very good idea
- Calculating belief state for future time-step works better
- Changing what the Agent prioritises – Chasing the Prey or Running from the predator – also changes the results and running away from the predator seems to be a good approach as the agent seems to stumble onto the prey while doing so.

Defective Survey Drone

- We can say that when the survey drone is defective, we get reduced survivability as a result of false negatives
- There is only a slight difference since the defective drone passes on correct information most of the times.

AGENT-9 (Theoretical, as mentioned by Dr.Cowan in the Class)

- We can calculate utilities for the next state and move accordingly.
- The utilities could take into account the defective drone’s probabilities of giving a false negative (0.1)