CITS3001 Project
2022

Red vs Blue Political Simulator

Student 1	Student 2
Reiden Rufin | 22986337	Nathan Eden | 22960674

Introduction

The game itself is set in a fictitious country where two teams are seeking geopolitical influence over its population. The population of the country - the green team will either be voting or not voting, this will later represent the winner of the game. Two major teams - Red and Blue team, are trying to outclass and outsmart each other by sending messages that will hopefully influence the voting statuses of each member of the green population. The Red Team is an authoritarian state with radicalised ideologies that are similar to an upper-right government. On the other hand, the Blue Team is a democratic sector, maximising community freedom identical to a bottom-left government . To elaborate further, let us first take a look at how each team will run. The Red team's inherent goal is to stop as many of the green population from voting by the end of the game. On the other hand, the Blue team is simply the antithesis of this; in that they are trying to make as many green members vote by the end of the game. During the game, each team will take turns distributing their messages across the green population (see further below). This process will repeat until the Blue team eventually expels all of its energy, in which the game will end. The winner will be based on the total number of the voting population contrasted against the total number of the green population.

Assumptions

Let us first clarify assumptions that were discussed and implemented within the game.

Assumption 1 - **Uncertainty Interval**

Our design of the game creates a fixed scale of [0,1] for the uncertainty interval. In that sense, we are randomly generating uniform values for the uncertainty of each green agent. This is generated within the Game's constructor as seen below.

```
uncertainty = round(
     random.uniform(uncertainty_range[0], uncertainty_range[1]), 2
)
```

The range of uncertainty specifies the minimum and maximum values for how persuadable any said agent will be for the simulation. For example a green agent with an uncertainty value closer to the max integer within the range; let us say 0.8, will be more uncertain about their current opinion. In contrast, an agent with an uncertainty closer to the minimum value; let us say 0.3, will be more certain about their current opinion.

This would mean that the first green agent stated with the 0.8 uncertainty will be more likely to be influenced in comparison to the second green agent with the 0.3 uncertainty.

Assumption 2 - **Green Node Opinions**

The game will set an opinion value for each green agent. This will be set either to True or False. What this inherently represents is whether or not a green agent's opinion is currently for the Blue team or Red team respectively. A True statement will result in that green agent currently 'supporting' the blue team, and vice versa for False. As for assigning these values to each agent, our design takes in a user parameter -p {int}, where {int} represents the total initial voting population of the green agent. See the below bash command for example.

```
~$ python Game.py -ge 100 -gp 5 -gr 10 -u 0.0,1.0 -p 50
```

The final argument -p {int} where {int} is 50, will represent that 50% of the starting green nodes will initially have their vote statuses set to True at the game's creation. This is implemented within the Game's constructor below

```
while(agent_id < voting_pop):
      vote_status = True
      break
```

In retrospect, at the end of the game we will count the total number of each agent's vote_status where it is True and compare this against the given total number of green agents. The more vote_status that are True, the better for the Blue team and less will favour the Red team.

Assumption 3 - **Red / Blue Agent can affect Green Agent's opinion**

Our Red and Blue agent's change the green agents' opinion. This is done using the 'vote_status' variable within the code of our Red and Blue agents which was explained in assumption 2. This can be seen below:

Blue Agent Code:

```
if(self.will_vote_status_change(certainty)):
     Green_agent.vote_status = True
```

Red Agent Code:

```
if(self.will_vote_status_change(potency)):
      Green_agent.vote_status = False
```

The project spec states that the red agent has knowledge of the green agent's opinion. From this we assumed that this must also be the case for the blue agent (the blue agent knows the opinion of the green agents). Therefore because both agents have knowledge of the green agent's opinion we assumed that they also have the ability to change the green agent's opinion themselves.

Project Specification (scenario-updated-26Sept.pdf)

Q. Does the red/blue agent know exactly what opinions and uncertainties the green agents have? And the connections?
Answer: No, the red and blue agents do not know that
Q. Does the red agent know just the opinions of green agents?
Answer: Yes.

Assumption 4 - **Files cannot be read to generate a graph**

As reading the network file given to us was optional, we have decided to not implement this functionality. Instead we will be generating a graph at runtime, which will be discussed later (see Implementation of Agents). But to give a brief rundown of the graph generation, it can be seen below

```
for agent in self.green_team:
            for agent2 in self.green_team:
                if agent2.unique_id > agent.unique_id:
                    if random.randint(0, 100) <= edge_probability:
                        agent.connections.append(agent2.unique_id)
                        agent2.connections.append(agent.unique_id)
```

Assumption 5 - **Red / Blue Agent knows the messages of the other**

In order to ensure our agent's algorithm executes properly, they must also know the other agent's list of messages. They will know this in the below code within their respective minimaxes.

```
red_agent_messages = []
for messages in red_agent.messages:
        red_agent_messages.append(red_agent.messages[messages])
blue_agent_messages = []
```

```
for messages in blue_agent.messages:
        blue_agent_messages.append(blue_agent.messages[messages])
```

Assumption 6 - **Red's Messages are 'stronger' than Blue's Messages.**
As per the project specifications, it is evident that the Blue Agent is more favoured as it has the power to end the game as the only conditional is that the Blue Agent's energy level depletes to 0. Additionally, Red Agent is inherently disadvantaged as it has a consequence for sending messages, such that the longer the game goes the more it loses. But the Blue Agent's power to end the game is Overpowered.

For some context, the Blue Agent is a democratic power whilst the Red Agent is an authoritarian sector. This means that the Red Agent will be using underhanded techniques such as sending threatening messages. What this essentially means is that the Red Agent's uncertainty change per potent message is much higher than that of the Blue's. Additionally, the blue agent should have a much harder time as they are more 'peaceful' in the sense that they expose themselves as fear-mongering warlords.

With that in mind, we have chosen to increase the "strength" of the Red Agent in order to balance out the game, such that an equivalent 'power' is given to the Red Agent as they are not able to decide how to end the game.

Assumption 7 - **Grey Agent's Messages will be distributed to all Green Agents.**
We have decided that if the Grey Agent's affiliation lies with the Red Team, then the only downside is that no follower loss will occur during that specific round. As such, it would be necessary that if Grey Agent is "part" of the population, then they too should have the ability to interact with every Green Agent, regardless if they are no longer a follower of the Red Team.

Selection and Design of AI Technology

Methodology

Parameters that are hard coded

Energy Level
The energy level has been hard coded as well as the energy loss whose value depends on the message that is sent.

Messages for Red Agent/Blue Agent

Each message for the red/blue agent has a corresponding value for the certainty/potency, energy loss/follower loss and uncertainty change which is hardcoded.

This helps in making each message distinct from each other as each of them have different values for certainty, energy loss/follower loss and uncertainty change.

The energy loss/follower loss and uncertainty change are hard coded in a way that they scale appropriately depending on the certainty level/potency of the message. This means that message 5 in blue agent will have a higher certainty than say message 2, this means that it also has a higher uncertainty change and energy loss then message 2.

Parameters to be inputted

Inputted parameters that are needed are

- Total number of green agents
- Probability of connections between green agents
- Percentage of green_pop that are grey_agents
- Uncertainty range
    - E.g. (-u 0,1) will give us an uncertainty range between 0 and 1
- Percentage of green agents that want to vote initially

Types of methods used to make agents intelligent

For our intelligent implementation, we partially did research into reinforcement learning, but found that it was too complex to implement. As a result we decided on using the minimax algorithm with alpha/beta pruning for our agent's intelligence.

Both the red and blue agent use the minimax algorithm, allocating themselves as the maximising player while also simulating the minimising player by creating a hypothetical green team to send both the maximising and minimising player's messages (see assumption 6). Additionally, both agents will be running at a depth of 3. Further increases to the depth showed a significant increase in the time complexity of the program even when alpha/beta pruning is used. This implementation can be seen in the game's file under the following functions:

```
def red_agent_minimax(self, green_team, red_agent, depth,
maximizing_player, blue_agent, alpha, beta):
...
def blue_agent_minimax(self, green_team, blue_agent, depth,
maximizing_player, red_agent, grey_agent, alpha, beta):
```

Game Play

The game is a turn-based simulator that mimics gameplay similar to already existing games such as Twilight Struggle, SuperPower2, Democracy 4 to name a few. As such our project was modelled in a way that each team will carry out their turn after the other for each round. We have chosen for the red team to start first, followed by the blue team, then finally the green team. To go in detail, let us take a look at the execute() statement for the game.

```
while self.blue_agent.energy_level > 0:
      if(self.blue_agent.energy_level <= 0):
            break
      print("Starting Blue Energy: ", self.blue_agent.energy_level)
      total_voting = 0

      red_message = self.red_agent.send_message()
      total_follower_loss = 0
      ...
```

Here, the game will run until the blue team expels all of their energy. The red will begin first by extracting what message it will send to all green agents that it can currently communicate with. We will also keep track of the total voting population at the start of this round and the associated follower loss for the message distributed by the red team.

```
red_uncertainty_change, follower_loss =
self.red_agent.red_move(green_agent, red_message)
total_follower_loss += follower_loss

self.change_green_uncertainty(green_agent.uncertainty,
red_uncertainty_change)
```

Here the red agent will move first, collecting necessary information about the green agent within its own class and returning the associated uncertainty change for the green agent and how many followers it will lose after sending this message. Do note, that the uncertainty change here will be used for the green node's interaction with each other.

```
blue_message = self.blue_agent.send_message()
```

The blue team will then send its corresponding message, however they do have a lifeline of using a grey agent. Similarly for the red agent, it will perform its move but instead of returning a follower loss it will return the associated energy loss for sending that message.

```
uncertainty_change, energy_loss =
self.blue_agent.blue_move(green_agent, blue_message)
total_energy_loss += energy_loss

self.change_green_uncertainty(green_agent.uncertainty,
uncertainty_change)
```

Finally the green agent's will interact with each other. However, this interaction will only persist if an edge exists between any two green agents.

```
green_nodes_visited = []
for green_agent in self.green_team:
    if(green_agent.connections):
       for neighbour in green_agent.connections:
           if(neighbour > green_agent.unique_id):
              continue
           else:
              if((green_agent.unique_id, neighbour) not in
green_nodes_visited):

green_nodes_visited.append((green_agent.unique_id, neighbour))
self.green_interaction(green_agent, self.green_team[neighbour])
```

How this essentially works is it will only perform the interaction between any edge once and only once. As for how the interaction actually works let us take a look at how green nodes inherently interact with each other.

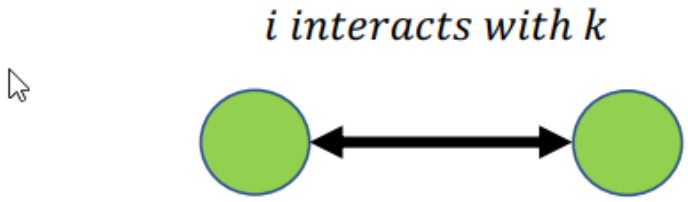
```
elif(green_agent.uncertainty > neighbor_node.uncertainty):
            new_uncertainty = abs(green_agent.uncertainty -
neighbor_node.uncertainty) * 0.125
```

```
            green_agent.vote_status = neighbor_node.vote_status
            green_agent.uncertainty -= new_uncertainty
            green_agent.uncertainty = round(green_agent.uncertainty,
2)
```

This simply executes the following diagram from the specifications of the project.

$x_i = 1$ and $u_i = 0.2$ (meaning i wants to vote)

$x_k = 0$ and $u_k = -0.2$. (meaning k does not want to vote)



Is $u_k < u_i$? Yes. Since uncertainty of k is less than that of i's, the new values will be as follows:

$x_i = 0$ and u_i = ? (i's opinion has changed to not vote, but you need to think of a clever way to have a new uncertainty value here)

$x_k = 0$ and $u_k = -0.2$ (nothing will change there)

The agent with the lower uncertainty will dominate. So this will cause a shift in the agent with the higher uncertainty's opinion to be that of the dominating agent. As such we will scale the uncertainty change needed by the absolute difference between the two uncertainties of the two agents being compared. We then take 1/8th of this value and subtract it from the green agent with the higher uncertainty. We can also reverse the entire code block for the case where the current visiting green_agent's uncertainty is less than its neighbour's uncertainty.

Red Agent Uncertainty Change

The red agent cannot directly change the uncertainty of the green agent as per the specifications of the project but they can, however, specify the direction the uncertainty will change towards. The red agent will have 10 levels of potent messaging which have

been hard coded as stated previously. At each level there is a set follower loss and the direction of uncertainty change associated with it

```
if message == self.messages[0] or message == self.messages[1]:
            potency = 0.2
            follower_loss = 0.15625
            uncertainty_change = 0.03125
        elif message == self.messages[2] or message ==
self.messages[3]:
            potency = 0.4
            follower_loss = 0.03125
            uncertainty_change = 0.0625
        elif message == self.messages[4] or message ==
self.messages[5]:
            potency = 0.6
            follower_loss = 0.00625
            uncertainty_change = 0.125
        elif message == self.messages[6] or message ==
self.messages[7]:
            potency = 0.8
            follower_loss = 0.0125
            uncertainty_change = 0.25
        elif message == self.messages[8] or message ==
self.messages[9]:
            potency = 1.0
            follower_loss = 0.025
            uncertainty_change = 0.5
```

Our implementation goes as follows. The messages at each level of potency, where there are five total potencies, are divided into two in order to generate 10 total messages. Each level's associated follower loss determines how many followers will be lost per agent that the red agent communicated to with the given message. In a hypothetical scenario where there are 100 green agents talking to the red agent for a round, if the red agent decides to distribute a highly potent message, that is, ones with potency 1.0 as shown in the code above; then they will lose 2.5% of green agents.

The uncertainty change is the direction at which the red agent will try to change the current interacting green agent's uncertainty to lean towards. Let us again use the

hypothetical scenario from above. In this scenario the uncertainty change will be 0.5 (see assumption number 6), which is a huge change considering our set uncertainty interval of [0,1].

This value is then utilised within the move for the red turn where, as we the agents do know the opinion of the green agent, it will want to increase the uncertainty of green agents whose vote statuses are currently going for the blue team. Hence, the red agent will check if the green agent's vote status is False. In which case, it would like to lower the uncertainty, and increase uncertainty if True. This can be seen below

```
if (green_agent.vote_status == False):
                uncertainty_change *= -1
```

We make this a negative for later when we look at how the green agent's uncertainty changes directly. The final step is changing the opinion (see assumption number 3). The associated potency from earlier is used here to generate a probability of changing the opinion.

```
def will_vote_status_change(self, potency):
     return random.randint(0, 100) <= potency * 100

def red_move(self, green_agent, message):
     ...
     if(self.will_vote_status_change(potency)):
           green_agent.vote_status = False
```

Using the hypothetical scenario where the potency is 1.0, this would then generate a 100% probability of changing the opinion. As any random integer between and including 0, 100 will always be less than or equal to 100.

Blue Agent Uncertainty Change

Looking at the blue agent, the implementation is very similar. The major key difference here is that instead of having a follower loss, it will have an associated energy loss.

```
elif message == self.messages[0]:
            certainty = 0.1
            energy_loss = 0.05
            uncertainty_change = 0.02
        elif message == self.messages[1]:
```

```
            certainty = 0.2
            energy_loss = 0.1
            uncertainty_change = 0.04
        elif message == self.messages[2]:
            certainty = 0.3
            energy_loss = 0.15
            uncertainty_change = 0.06
        elif message == self.messages[3]:
            certainty = 0.4
            energy_loss = 0.2
            uncertainty_change = 0.08
        elif message == self.messages[4]:
            certainty = 0.5
            energy_loss = 0.25
            uncertainty_change = 0.10
        elif message == self.messages[5]:
            certainty = 0.6
            energy_loss = 0.3
            uncertainty_change = 0.12
        elif message == self.messages[6]:
            certainty = 0.7
            energy_loss = 0.35
            uncertainty_change = 0.14
        elif message == self.messages[7]:
            certainty = 0.8
            energy_loss = 0.4
            uncertainty_change = 0.16
        elif message == self.messages[8]:
            certainty = 0.9
            energy_loss = 0.45
            uncertainty_change = 0.18
        elif message == self.messages[9]:
            certainty = 1
            energy_loss = 0.5
            uncertainty_change = 0.2
        return [certainty, energy_loss, uncertainty_change]
```

This is essentially the same as the follower loss implementation. Within the hypothetical scenario that the blue agent sends the highest correction message with the certainty of

1.0. The associated energy loss will be 0.5 per green agent that it interacted with. But the case of the blue agent is that they can always interact with every green agent. Another difference to look for here is the fact that the uncertainty change for the blue messages are significantly lower than that of the Red Agent's, this was explained in assumption 6. Likewise to the red agent again, it will know the exact opinion of the green agent it is currently talking to. Hence, it will check if the current opinion is associated with True, and set the uncertainty change to be negative. A very similar approach is then seen with the blue agent for the opinion change of the green agent, but we use the certainty associated with the message rather than the potency. This can be seen below

```
def will_vote_status_change(self, certainty):
        return random.randint(0, 200) <= certainty * 100
```

Going back to assumption 6, we need to make the blue agent have a harder time influencing the green population as they voluntarily choose to not use underhanded fear tactics as part of their campaign.

But why exactly have we been making these negatives?
The answer is because of how the green agent's uncertainty change will work.

Looking back at the game, we will return the values of both the blue and red uncertainty changes as part of their moves. This will then be used for the following function.

```
def change_green_uncertainty(self, green_agent_uncertainty,
uncertainty_change):
        green_agent_uncertainty += uncertainty_change
        if green_agent_uncertainty > self.upper_limit:
            green_agent_uncertainty = self.upper_limit
        elif green_agent_uncertainty < self.lower_limit:
            green_agent_uncertainty = self.lower_limit
```

This will simply change the green agent's uncertainty to be the addition of its current uncertainty and the given red or blue uncertainty change. This is where the negative that we created from earlier will be important. If the green agent's current status is False, and the red interacts with this, it will want to decrease the uncertainty of this agent. Hence, adding a negative value will just subtract the current agent's uncertainty, making it lean towards certainty. A check we implemented was introducing the upper and lower limits of the range. As our uncertainty interval uses [0,1], we will run into

scenarios where we will overflow this given range. So to counteract this, we just set the uncertainty to the closest upper and lower limits of the range if it overflows after the calculation.

Implementation of the Agents

Project Runtime

For this project, we have simply used an array to store the necessary information. This encompasses multiple data such as the green team which is an array of Green Agent objects, the connections per each Green Agent to build the network, the grey team which is another array of Grey Agent objects.

A major bottleneck to this program is that it is heavily dependent on the size of the green network, as both agents including the green agents themselves will have to iterate over all their respective connections across the green network. Furthermore, when we create the undirected graph we will have to iterate over the green network twice, generating a quadratic time complexity for the constructor of the game. To find our total runtime we can simply use the time module, to calculate the total program runtime. We also assume no user inputs for these tests, such that both agents will immediately start playing against each other.

Our tests are as follows.

Parameters of 100 green nodes, 5% edge probability.

```
took 3.4361073970794678 to run
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

With these parameters, it took a total of 3.4 seconds

Parameters of 100 green nodes, 50% edge probability

```
took 7.214942455291748 to run
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

As we can see, although the green nodes are consistent with the first test, increasing the edge probability significantly caused an increase with the total runtime. This is due to how there are more green nodes that can talk to each other, which means that the green interaction will take longer.

Parameters of 200 green nodes, 5% edge probability

```
took 8.09951663017273 to run
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

As we can see, the total runtime has more than doubled to 8.10 as we have also doubled the number of nodes. This is because there are more green nodes that both the agent's will have to talk to per round.

Parameters of 200 green nodes, 50% edge probability

```
took 22.66788387298584 to run
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

Further increasing the number of nodes with a higher edge probability gave us a runtime of 22.67 seconds. From these tests we can conclude that the total runtime of the project will be dependent on both the inputted number of green agents and the edge probability for the green network.

Language, Approach and Libraries

We have chosen to use Python for this project, mainly running on version 3.10.x. We also decided to use an object-oriented approach. This is mainly as we want to reuse multiple data, especially with the green agent as they each should have their own uncertainty, connections, etc. Hence data encapsulation was used in a way to organise and create multiple green agents with different sets of information.

For the libraries we have used the following below

```
import networkx as nx
import matplotlib.pyplot as plt
import prettytable as pt
import random
import sys
import copy
import math
import time
```

All of this can be installed with a $pip command.

Running the Game

The game uses multiple command lines in order to be executed. Please see below.

```
~$ python Game.py -ge 100 -gp 5 -gr 10 -u 0.0,1.0 -p 50
```

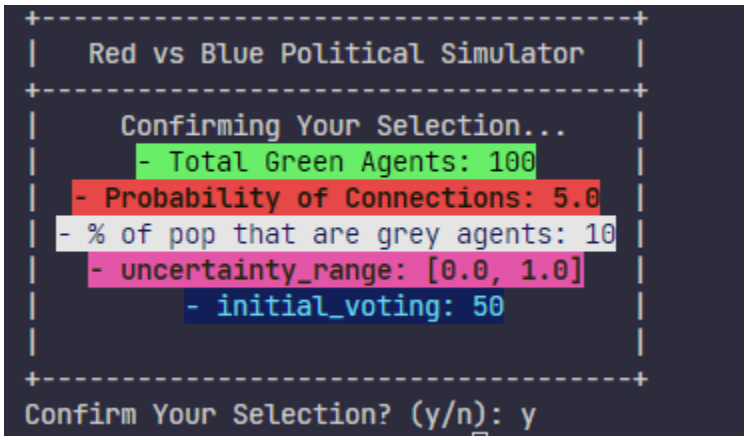
Let us break down this execution.

It will first pass in the file for which the Game is executed (Game.py)
The first argument -ge [x] represents the total number of green agents that will be in the game. In this case it will be 100.
The second argument -gp[x] represents the probability of connections between green nodes. In this case it has a 5% connection.
The third argument -gr[x] represents the percentage of grey agents of the population. In this case 10% of the total green population are grey agents.
The fourth argument -u [x,y] represents the inclusive uncertainty interval. In this case the uncertainty interval will be between and including 0.0 and 1.0.
The fifth argument -p[x] represents the initial voting population. In this case 50% of the green population will be voting at the game start.

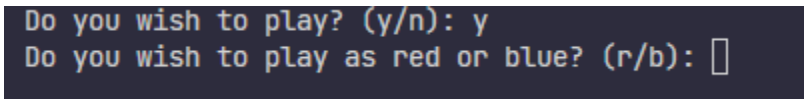
What this means, is that under these commands the game will execute with

- A total of 90 green agents (remember, grey agents take up a portion)
- A 5% connection between the green agents
- 10% = 10 grey agents
- Each green agent's uncertainty will be between the inclusive range of 0.0 and 1.0
- 45 of the green agents will initially have their vote statuses set to True.

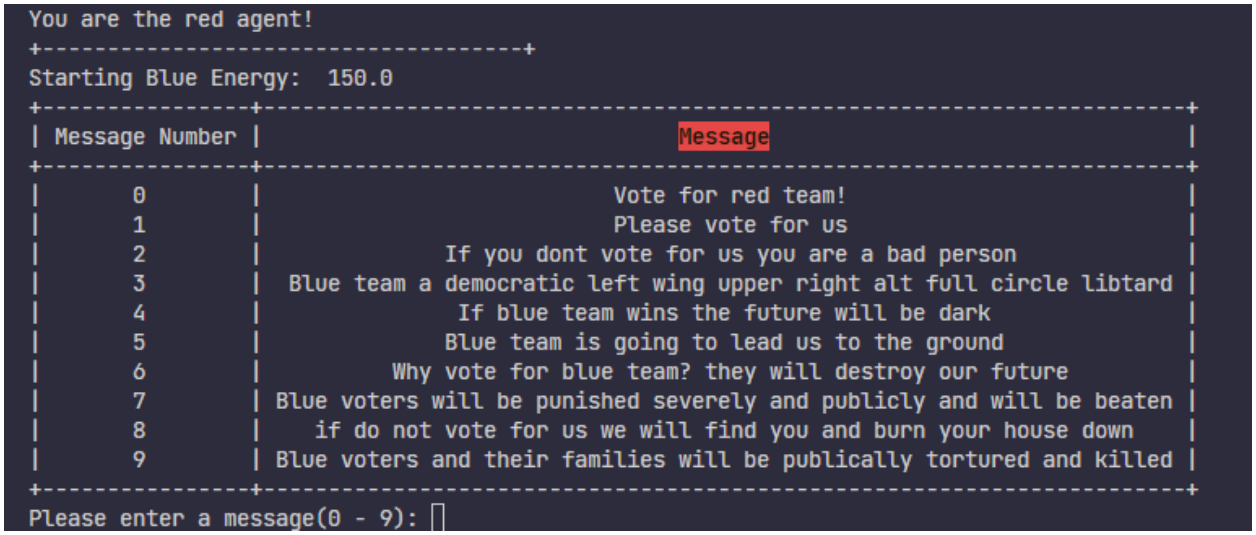
When run, the user will be greeted with a confirmation of their inputs.



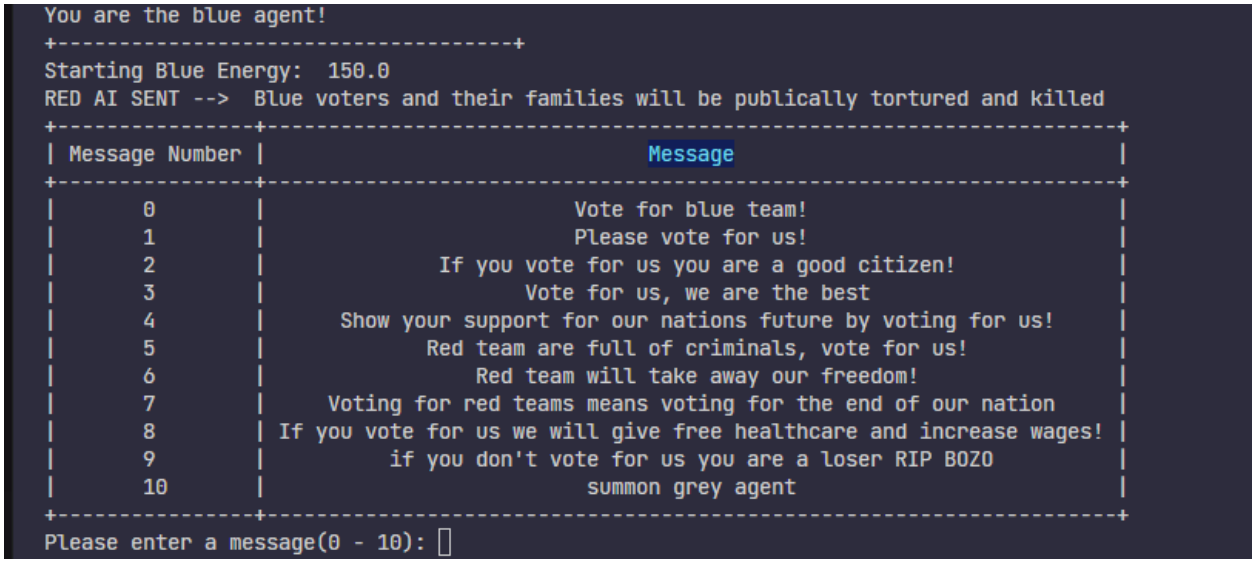
If they decide to continue, they will be asked if they would like to play. If they do decide to play, they are then asked which agent they would like to play as (only limited to Red or Blue)



Finally, at each game execution they are given a list of messages they could potentially choose from. For example playing as a red agent below will yield:



And likewise for the blue agent, they are given their list of correction messages with the addition of choosing a grey agent if they exist within the game.



As a user you are given the total population, total voting population, total red followers, and total blue energy at every iteration of the game.

```
---------------------------------
Total Population: 90
Total Voting Population:  0
Total Red Followers: 88
---------------------------------
================= NEXT ROUND ==================

Starting Blue Energy:  150.0
```

Agent Design

Green Agents

The green agents are put into an array where they each have a corresponding unique identifier, their associated green connections, their current vote status, their current uncertainty and whether or not they are able to communicate with the red team.

```
class green_agent:
    connections = []
    communicate = None
    def __init__(self, connections, unique_id, vote_status,
uncertainty):
        self.connections = connections
        self.unique_id = unique_id
        self.vote_status = vote_status
        self.uncertainty = uncertainty
        self.communicate = True
```

This then creates a team of grey agent's with their own values. The creation can be seen in the game from here where their connections, id, current vote_status and associated uncertainty are given to them.

```
for agent_id in range(int(new_green_total)):
            vote_status = False
            uncertainty = round(random.uniform(uncertainty_range[0],
uncertainty_range[1]), 2)
            connections = []
            while(agent_id < voting_pop):
                vote_status = True
```

```
                break

self.green_team.append(green_agent.green_agent(connections, agent_id,
vote_status, uncertainty))
```

For the green team we have decided to use a static network which is generated at runtime. This can be seen from here.

```
for agent in self.green_team:
            for agent2 in self.green_team:
                if agent2.unique_id > agent.unique_id:
                    if random.randint(0, 100) <= edge_probability:
                        agent.connections.append(agent2.unique_id)
                        agent2.connections.append(agent.unique_id)
```

This essentially creates an undirected graph where given the edge_probability from the user parameter earlier, creates an edge between any two green nodes. The unique identifier of the neighbouring green agent is then given to the current green agent's list of connections, hence generating an edge between them.

As for the properties of this network, it is simply an undirected, unweighted graph. In this case, links cannot be added or removed during the play, as it is created statically with no option for complex functionalities.

Red and Blue Agent

The Red and Blue Agents are the major players of the games, as such they should be encapsulated into their own class for easier abstraction.

Message Potency

The messages (1-5) for the red agent and (1-10) for the blue agent each have a unique message potency/message certainty that scales the higher the number of the message. I.e for the red agent message 1 has a lower potency than message 2 and for the blue agent message 2 has a lower certainty than message 4. The higher the potency (red agent) or certainty (blue agent) the more the message will affect the green node's uncertainty, the higher the chance of changing the green node's opinion but also the higher the follower loss/energy loss is. For each message the potency/certainty, energy

loss/follower loss and the uncertainty change has been hard coded within the red and blue agent.

Red Agent Follower Case

The way that the follower numbers change in the red agent begins with a variable called follower_loss. This value of follower_loss depends on the potency of the message, a higher potency message will incur a higher follower loss. A counter variable will accumulate the total follower loss per green agent, so the total amount of follower loss will be equal to the value of follower_loss * the number of green agents the red agent interacts with on a given turn. The final number will then be used to specify an amount of followers the red agent will lose i.e the number of green nodes red agent will no longer be able to interact with for future turns. This is seen below within the Game.

```
index = 0
            while(index < round(total_follower_loss)):
                green_agent = random.choice(self.green_team)
                if(green_agent.communicate):
                    green_agent.communicate = False
                    self.red_agent.followers -= 1
                    index += 1
```

As the red move returns the total follower loss, we can simply use this to count the follower loss per round and then set green agent's ability to communicate with the red agent to False, depending on how many followers are being lost per round.

Blue Agent Energy Loss

The energy level for blue agents begins with a hard coded value energy_level. The amount of energy the blue agent will lose per green agent depends on the certainty of the message that is chosen. A higher certainty incurs a higher energy loss. Each message has a hard coded value energy_loss. This value becomes subtracted from the blue agent for each green agent it interacts with. The total amount of energy loss in a given turn is equal to the value of energy_loss * the number of green agents the blue agent interacts with. This is shown below.

```
self.blue_agent.energy_level - total_energy_loss
            print("Blue Energy Level: ",
self.blue_agent.energy_level)
```

Grey Agents

Grey Agents are initialised nearly the same way as the Green Agents, such that they are put into an array indicating the “Grey Team”. They are also initialised with an id to uniquely identify them and their team, which indicates whether they are affiliated with the Red or Blue Team.

The grey agent first checks which team it is on, it will be passed a string which will be either, “Red” or “Blue”. The grey agent then proceeds to function exactly as a red or blue agent would except for the follower loss or energy loss respectively.

Validation of Agents

Simulations

For the simulations we used a shell script to automate the Game, where every simulation below we decided for the AI’s to play against each other. Additionally, each simulation was ran 100 times to ensure a moderate sample size for analysis, along with the results of each game such as the winner, the total voting population at the end of the game, and the total population of the game itself were written to a text file.

```
echo "Running The Game..."
for ((i=1; i<=100; i++))
do
    python Game.py -ge 100 -gp 10 -gr 10 -u 0.0,1.0 -p 0
done
```

What the goal here is to essentially check for fair play and that both agents are performing to the best of their abilities. To elaborate, this would mean that both agents should have close to a 50% win rate for each simulation, as this would mean that they are sending the best possible message each round for each game.

Simulation 1

The first simulation we will be using the below parameters

```
-ge 100 -gp 10 -gr 10 -u 0.0,1.0 -p 0
```

To clarify, this will use a total of 100 green nodes, an edge probability of 10%, 10% grey agent population, an uncertainty range of 0.0 to 1.0, and the initial voting population set to 0%

The results of this simulation are as follows:

```
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project> python .\result_analyser.py
Blue: 49
Red: 50
Tie: 1
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

We can see that the blue agent won 49 out of 100 games, the Red agent won 50 times out of 100 games and a Tie was only counted once. This gives us a win rate of 49% for the blue and a win rate of 50% for the Red.

This simulation also ran for about 1 - 2 minutes in total.

Simulation 2

The second simulation we will be using the below parameters

```
-ge 250 -gp 50 -gr 0 -u 0.0,1.0 -p 50
```

This will use a total of 250 green nodes, an edge probability of 50%, 0% grey agents, an uncertainty of 0.0 to 1.0, and an initial voting population of 50%. The results are below

```
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project> python .\result_analyser.py
Blue: 50
Red: 49
Tie: 1
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

Surprisingly, the win rate is nearly the same as first the simulation, albeit one extra win for Blue rather than red. What is most notable is that this simulation did in fact take around 20 minutes to run.

Simulation 3

The third simulation we will be using the below parameters.

```
-ge 100 -gp 10 -gr 10 -u 0.0,10.0 -p 0
```

For this simulation we will be using identical parameters of the first simulation. However, the major difference would be a broader uncertainty range of 0.0 to 10.0, rather than the tight 0.0 to 1.0. The results are below

```
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project> python .\result_analyser.py
Blue: 45
Red: 54
Tie: 1
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

We once again get 1 tie, however this results in Red having a higher win rate than it's predecessors. This simulation also took around 2 minutes to run.

Simulation 4

The fourth simulation we will be using the below parameters

```
-ge 150 -gp 100 -gr 0 -u 0.0,5.0 -p 50
```

We will be analysing the results of having 150 green nodes with a 100% probability of connections, this would mean every green node can talk to every other green node. We will also be testing its performance with 0 grey agents, a moderate uncertainty interval of 0.0 to 5.0, and an initial voting population of 50%. This results in the following

```
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project> python .\result_analyser.py
Blue: 40
Red: 58
Tie: 2
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

Red has gained a higher win rate once again. What this suggests is that, the more green nodes that can talk to each other, the potent message that red sends will be distributed across more nodes despite losing followers. This is because as each node has the inherent ability to communicate, it's vote status, despite not being able to communicate with the red agent anymore, still lingers within the network.

Simulation 5

For this simulation we are interested in seeing the results of a very tight uncertainty interval. That is, we will be using the same parameters as simulation 1 but with 0.0 to 0.5. The parameters are as follows

```
-ge 100 -gp 10 -gr 10 -u 0.0,0.5 -p 0
```

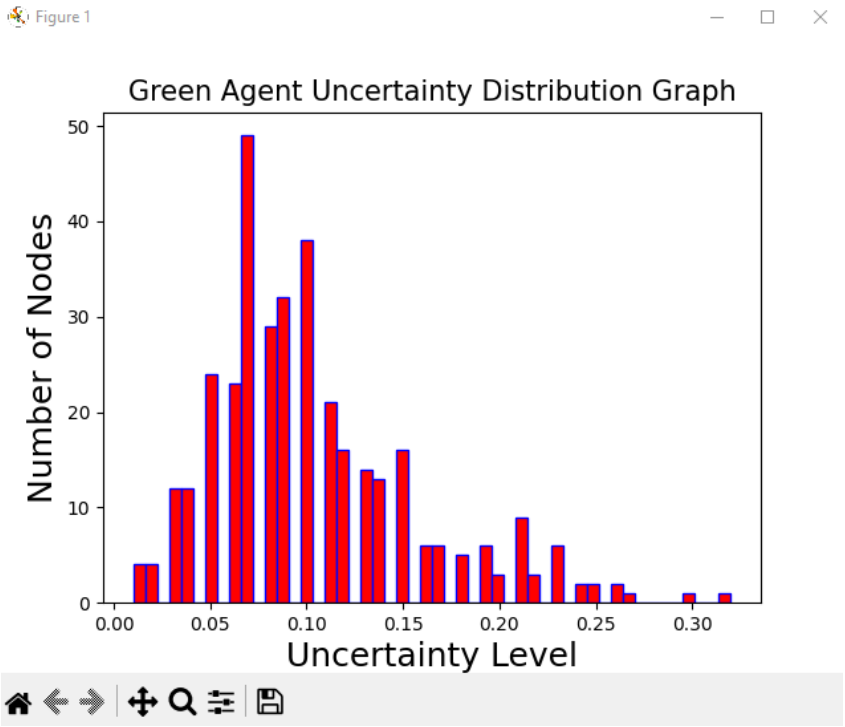
```
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project> python .\result_analyser.py
Blue: 57
Red: 42
Tie: 1
PS C:\Users\reide\Desktop\uni\CITS3001\CITS3001-Project>
```

Running this simulation resulted in Blue winning 57% of the games with at least 1 Tie still present and Red only winning 42% of the games.

Uncertainty Interval Effects

Effects of a Tight Uncertainty Interval

For a tight uncertainty interval, we will further look at Simulation 5 above. The uncertainty plot distribution are as follows

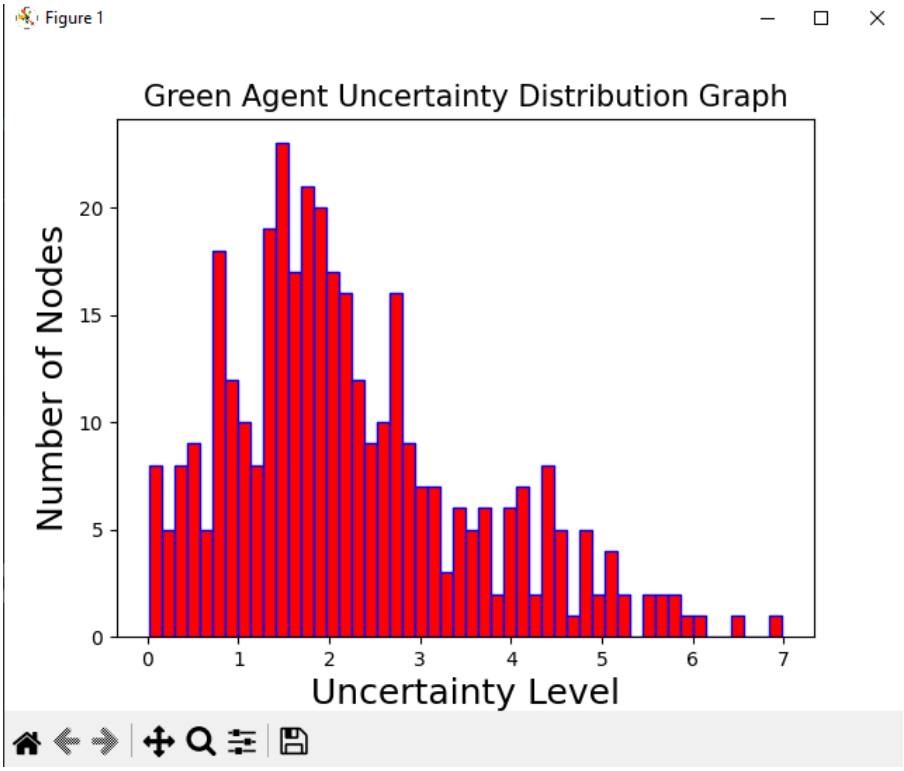


After smoothing out the graph by aggregating the uncertainties list to read in, we eventually end up with a positively skewed graph. This means that after the end of the simulation, the majority of green agents have higher levels of uncertainty at lower ranges, and as uncertainty rises past this point the trend becomes negative. Meaning the more uncertainty grows the less agents that have it. This corresponds to how as the game goes on for longer, the more “certain” an agent becomes after it talks to a red/blue/green agent.

The Blue Team also won a majority of the games with a 57% win rate. This suggests that it became harder for the Red Team to influence the green population, as a tight uncertainty interval yields less “uncertain” agents to be coerced.

Effects of a Broad Uncertainty Interval

For a broad uncertainty interval, we will further look at Simulation 3 above. Our plot distribution for a broad interval are as follows



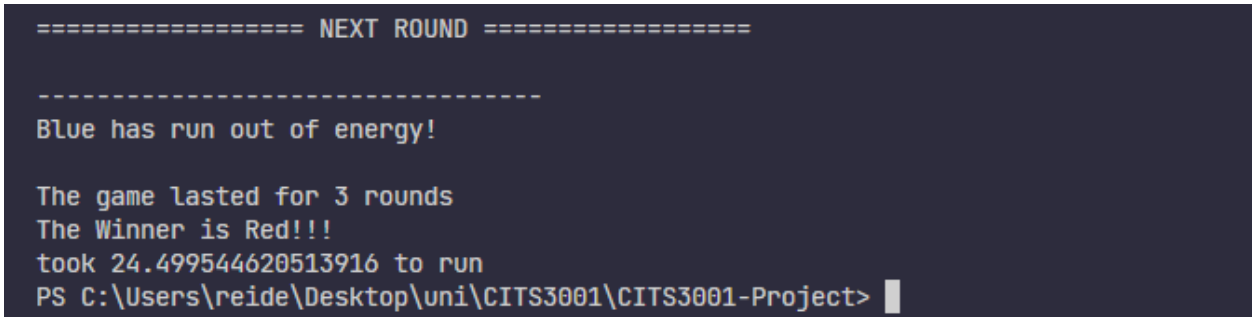
The following result after 4 total rounds are also positively skewed, albeit with a greater range. A good thing to note for the effects this has, is that it resulted in the Red Team winning more games. What this suggested is that a broader interval also gave rise to more “uncertain” green agents to be influenced. This suggested that it became easier for the Red Team to influence more agents.

Red Agent Strategy

The best strategy for the Red Agent to win the game is to mirror the Blue Agent’s strategy. To elaborate, if the Blue Agent plays like a blitzkrieg attack by rushing the game then it must also play aggressively. A good thing to note is with Simulation 3 and Simulation 4. With simulation 3, the Red Agent has a higher chance of winning when a broader uncertainty interval is introduced, as this makes more “uncertain” agents that could be influenced. With simulation 4, the Red Agent has another higher win rate. This is due to how a greater edge probability will result in more green nodes having the ability to talk to each other. As a result, despite losing followers, the Red Agent’s message and uncertainty changes could still persist across the green network.

In order for the Red Agent to win, it must also rely on the hope that the Blue Agent will send a Red-Affiliated Grey Agent, this will significantly boost its messages without the

drawback. Additionally if we run the parameters from Simulation 4, it seems as if the Red Agent is winning within a short number of rounds

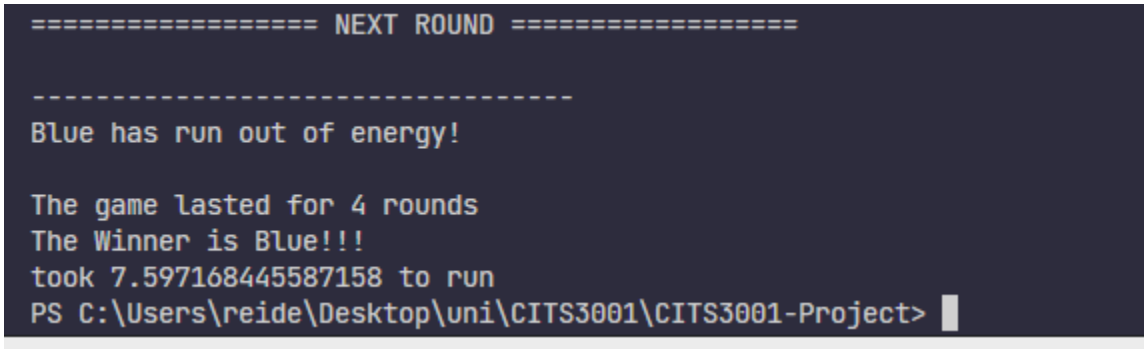


This was achieved when the Red Agent sent it's most potent message whenever the Blue Team did so as well.

Blue Agent Strategy

The best strategy for the Blue Agent to win the game is also incorporating a hyper-aggressive manoeuvre. The Blue Agent knows that it can end the game, and only it can do so. This resulted in our Blue Agent always sending out the highest "certain" messages associated with the highest associated energy loss. A good thing to note is with Simulation 5, with how a tighter uncertainty range made it easier for the Blue Agent to win.

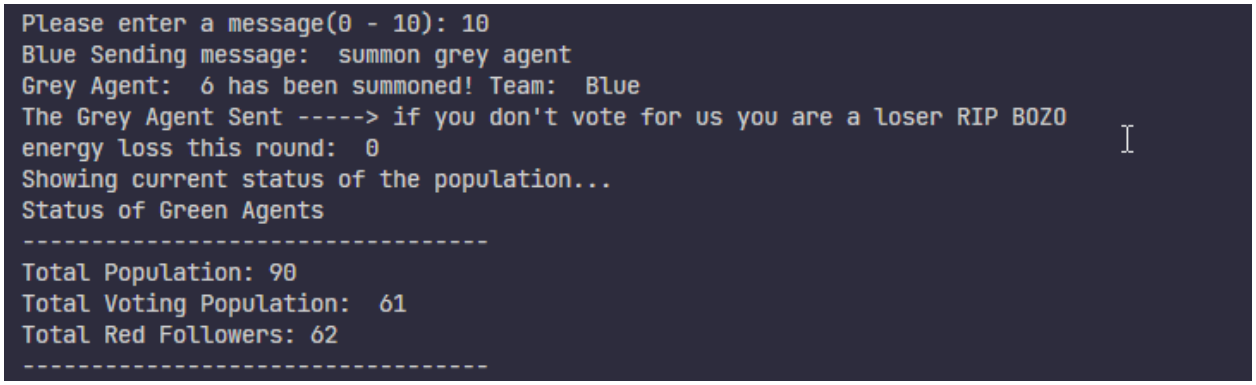
Using the same parameters as Simulation 5, it seems as if the Blue Agent wins at around 4 total rounds.



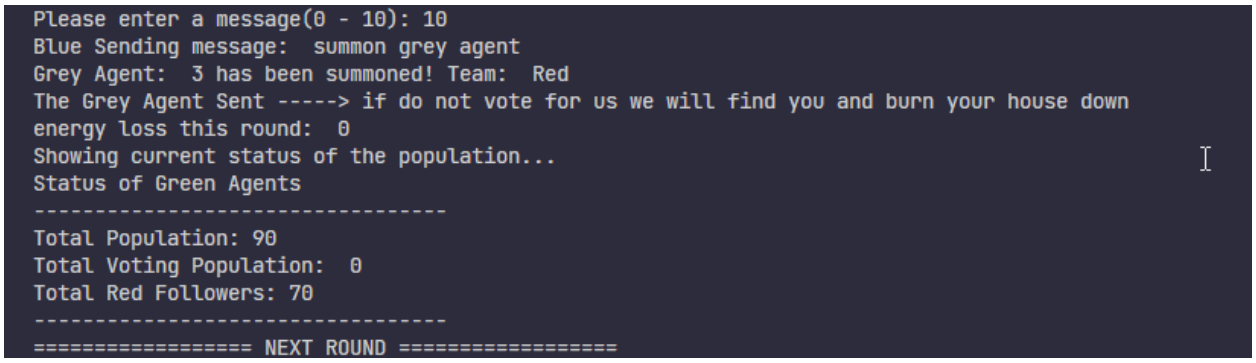
This further proves the hyper-aggressive strategy that it utilises by ending the game as quickly as possible. However, it did not seem to play defensively despite in rounds where it was at a significant disadvantage.

Effects of a Grey Agent

The Grey Agent only has a significant impact when it is affiliated with Red. Here we will test the effects of summoning a Grey Agent.



In this figure, we can see how a Blue-Affiliated Grey Agent sends a moderate level Blue Agent message, resulting in a total voting population of 61 out of the total 90 for that current round.



However, after sending another Grey Agent again that is Red-Affiliated, it sent one of the most potent messages from the Red Team, resulting in a total voting population of 0 as the Grey Agent can talk to every Green Agent, regardless if they are still a follower (see assumption 7). This was tested with the same parameters as Simulation 5, as we wanted to test the effects of a Red-Affiliated Grey Agent when put into an environment that favours the Blue Agent.

What is concerning is that, because Grey Agent does not have a drawback when sided for either team, it relatively becomes easy to recover from the “loss” of being affiliated with Red when the Blue Agent summons a Grey Agent. The 0 consequences of having no energy loss, means that the Blue Agent could just repeatedly exhaust their Grey Agent supply, forcing the Red Agent to lose followers per round as they must send at least one message per game.

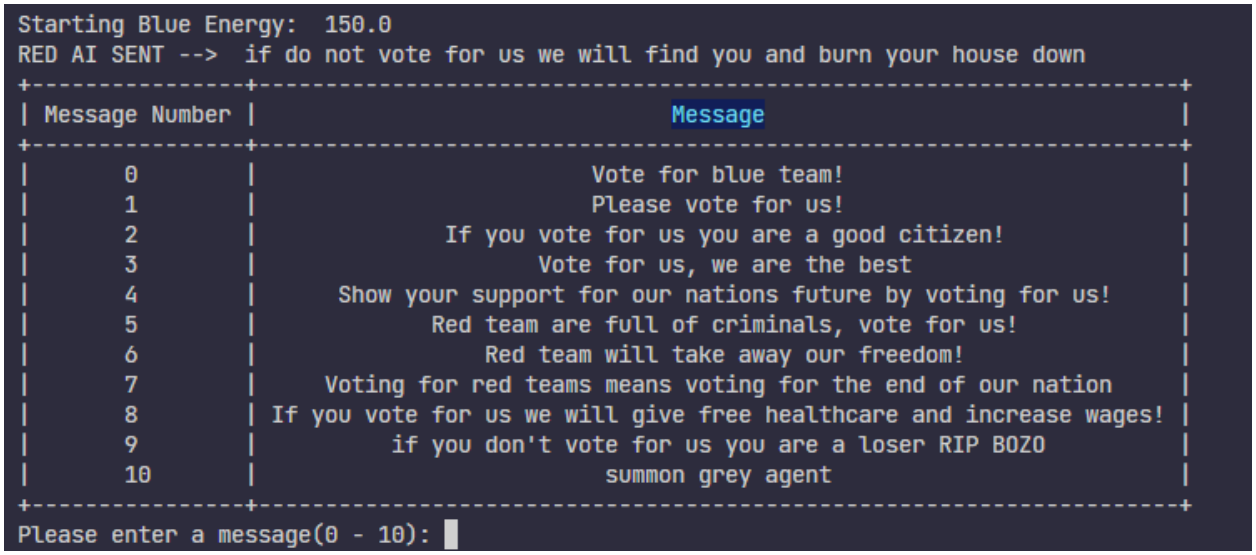
Performance of PvE (Player vs Agent)

For the following performance analysis of Player vs Environment, we will be running with the below parameters

```
-ge 100 -gp 50 -gr 10 -u 0.0,1.0 -p 50
```

When playing as the Red Agent, a human's initial strategy will be to just ignore the follower downside and send the most potent message per round. What we found was that the Blue Agent (the AI in this scenario), was also sending its two most certain messages per round. This was also the case when a human incorporates just sending the least potent message as the Red Agent. This suggests that the Blue Agent knows that it has the power to end the game despite whatever message we are saying. It also knows that sending the highest certain message will yield the greatest value. Which means, it is challenging to beat the Blue Agent when playing as a Red Agent.

When playing as the Blue Agent, an interesting find was that the Red Agent will start off with one of its two most potent messages.

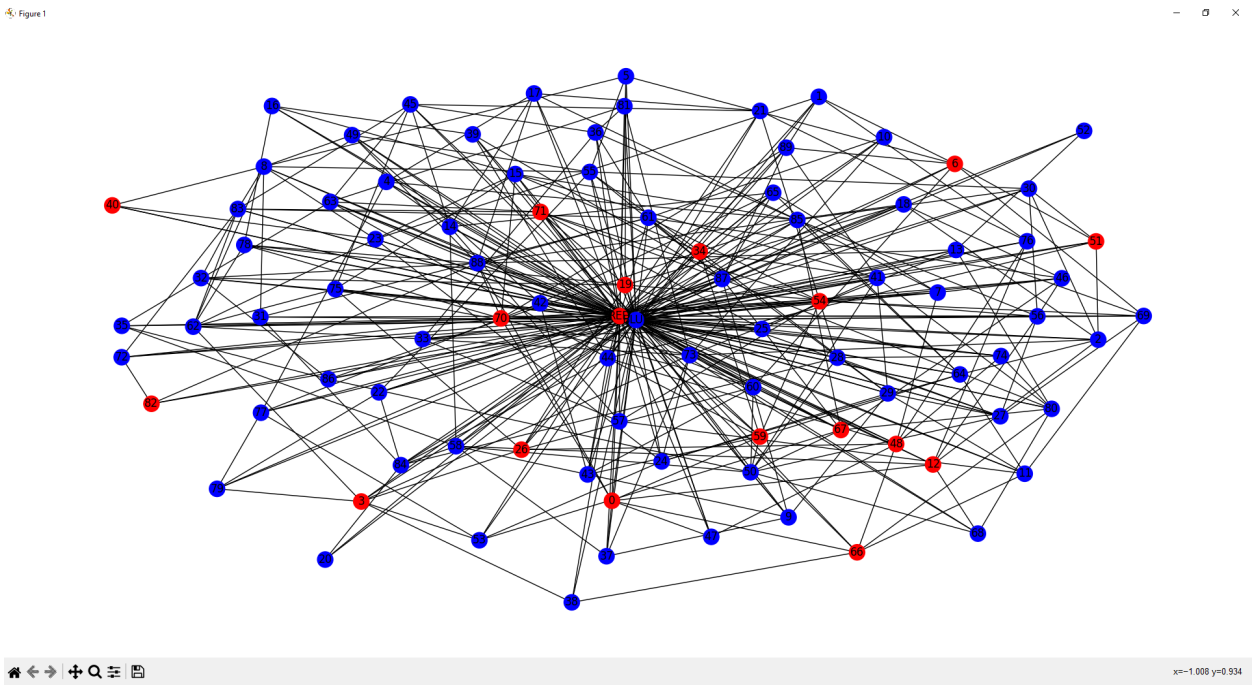


This Red Agent's strategy seemed to continue, suggesting that our Agent's were very aggressive and incorporated an offensive strategy. Playing against a Red Agent was least challenging in comparison to a Blue Agent. This is because of the rules specified as to how the game ends. Incorporating a hyper-aggressive strategy as the Blue Agent usually leads to a higher win rate. An interesting find is that, if we wanted to play hyper-aggressive while minimising loss such as summoning as many grey agents as possible, there were key downsides using this approach. Due to us, the player, not knowing the affiliation of the grey agent, if said grey was sided with the Red, they will also send the most potent message. However, when this occurred it was fairly easy to

stage a comeback by simply sending the most certain message and forgetting the Grey Agent strategy.

Visualisation

For the libraries used in this section, please refer to the Implementation of Agents header.



Above is what we will use to discuss this section of the report. Please refer to this figure when reading.

For visualisation we have incorporated a graph that is generated at runtime and displayed to the user at the end of each round. Furthermore, they are also displayed as a plot distribution for the uncertainties of the green population.

Parameters Used For Graph Visualisation

```
-ge 100 -gp 5 -gr 10 -u 0.0,1.0 -p 50
```

Colours and labels

Red Agent is labelled, ‘RED’ and is position near the centre

Blue Agent is labelled, ‘BLUE’ and is positioned near the centre
Green Agents are all the other nodes

- The number on the green agent nodes represent that specific green agents unique identifier
- If they are Red in colour it means that they are NOT VOTING
- If they are Blue in colour it means that they ARE VOTING
- 

    - This node denotes a green agent whose unique ID is 4 and IS NOT voting
- 

    - This node denotes a green agent whose unique ID is 8 and IS voting

Red Agents followers

The red agents followers are denoted by an edge connecting a node with the Red Agents node; over time, overtime as the red agent loses followers the number of edges connecting to the red agent will decrease. Therefore, the number of edges a red agent node has connected to will be equal to the number of green agents who are followers of red.

Blue Agent followers

The blue agent will be able to interact with every green agent and thus an edge exists between the blue agent node and every green agent node. Therefore, the number of edges a blue agent node has connected to it will be equal to the number of green agent nodes

Green Agents neighbours

A green agent has a neighbour for each other green agent it can talk to. For each of the neighbours per green agent node, there will be an edge that connects these nodes. For example if node 89 has a set of neighbours {88, 87, 86}, an edge will exist from 89 to each neighbour in the set.

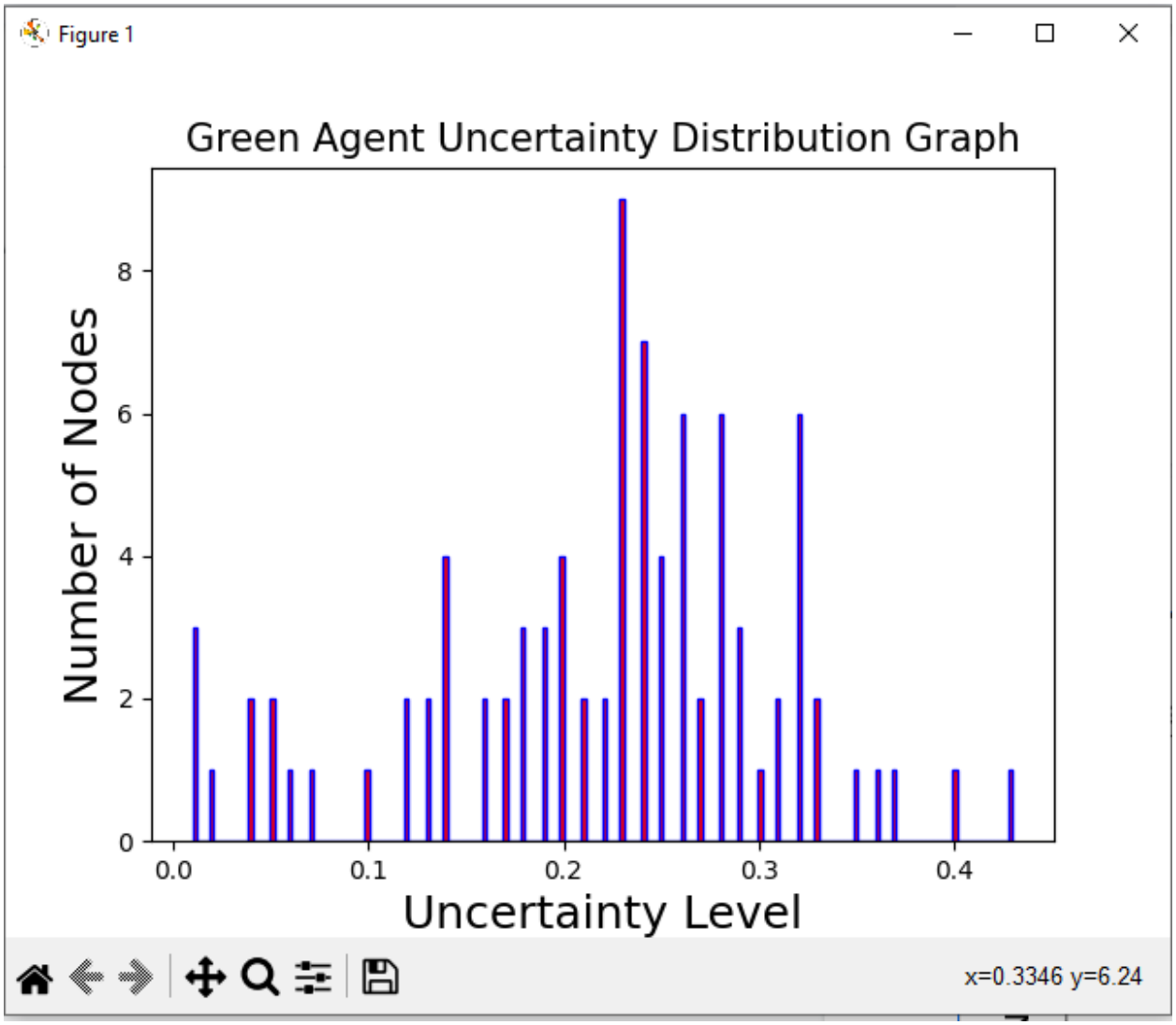
The number of edges a single green node will have:

- +1 if it is a follower of red (edge between itself and Red Agents node)
- +1 to blue agent node (edge between itself and the Blue Agents node)
- +x where x is the number of neighbours it has (edge between itself and x other green nodes)

Distribution of Green Agent Uncertainties

Parameters Used For Plot Distribution Visualisation

```
-ge 100 -gp 30 -gr 10 -u 0.0,1.0 -p 20
```



Above is what we will use to discuss this section of the report. Please refer to this figure when reading. Note this graph is only in relation to green agents i.e all of its data is derived from (all) the green agents.

Explaining the graph

The Y-axis (Number of Nodes) denotes the number of green agents

The X-axis (Uncertainty Level) denotes the level of uncertainty which ranges from 0.0 to the nearest 0.05 above the highest uncertainty level among the green agents (if the highest uncertainty level is 0.77 the graph will have a range of 0.8 (0.0 to 0.80))

Therefore, each bar will represent a specific uncertainty level and the number of green agents which possess that specific uncertainty level provided such uncertainty level exists among the green agents. For example in the above figure, there are 4 green agents who have an uncertainty level of 0.2.