6CCS3PRJ Final Year Project

Empirical Evaluation of Nash Equilibria Found in Non-Decreasing Congestion Games with Load-Dependent Failures

Final Project Report

Author: Hiro Funatsuka

Supervisor: Dr Maria Polukarov

Student ID: 1868443

Programme of Study: BSc Computer Science

April 8, 2022

Abstract

Non-decreasing congestion games with load-dependent failures (CGLFs) are a class of games that are proven to possess a pure strategy Nash equilibrium. The aim of this project is to conduct an empirical evaluation of data about the ratio between a Nash equilibrium and the optimal solution in CGLFs by relating to the price of anarchy and the price of stability. As well as the evaluation, the project provides specifications and implemented systems to find such an equilibrium and the optimal solution in the game.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Hiro Funatsuka

April 8, 2022

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr Maria Polukarov, for supporting and providing me helpful advice throughout this project.

Contents

1 Introduction **3**

1.1 Motivation . . . 3

1.2 Objective . . . 4

1.3 Report Structure . . . 4

2 Background **5**

2.1 Literature Review . . . 5

3 Specification & Requirement **9**

3.1 Specification . . . 9

3.2 Requirement . . . 12

4 Design & Implementation **16**

4.1 Preparation . . . 16

4.2 Building CGLF . . . 17

4.3 Building Nash Equilibrium Finder . . . 19

4.4 Software Testing . . . 22

5 Legal, Social, Ethical and Professional Issues **24**

5.1 Ethical Concerns . . . 24

5.2 British Computer Society Code of Conduct and Code of Good Practice . . . 25

6 Results/Evaluation **26**

6.1 Limitations . . . 26

6.2 Results . . . 28

7 Conclusion and Future Work **32**

Bibliography . . . 34

A Extra Information **35**

A.1 Tables . . . 35

B User Guide **37**

B.1 Instructions . . . 37

C Source Code **39**
C.1 player.py 40
C.2 resource.py 41
C.3 strategy_ profile.py 42
C.4 equilibrium.py 43
C.5 cglf.py 44
C.6 main.py 45
C.7 test_ strategy_ profile.py 46
C.8 test_ cglf.py 47
C.9 test_ equilibrium.py 48

Chapter 1

Introduction

1.1 Motivation

World Health Organization officially declared COVID-19 as a pandemic on 11 March 2020 [6]. This pandemic has changed the way humans behave. For example, many countries have employed lockdown or equivalent measures to it in order to restrict the population flow that potentially causes coronavirus clusters. As a result of these measures, citizens are prohibited or refrained from going out, especially to commercial and recreational facilities.

Some counties that cannot employ lockdown have faced an issue that some citizens and organisations ignore the restrictions that are not legally valid. For example, Japan is a country that cannot employ lockdown, and therefore, had to declare a state of emergency as an alternative measure to it. The only punishment based on the newly revised law under the emergency was to impose a fine on facilities that ignore the request for reduced business hours from the government, and there was no punishment for individuals unless they disobey the laws. However, this measure to facilities finally appeared to be less effective since there were facilities whose operating income produces higher profit than following the request even though they had to pay the fine. This event can be seen as a player's strategic decision making in a non-cooperative game taking account of the other player's strategies. Another impact of the pandemic on human behaviours was overbuying. Because of fake information saying potential shortage in some products as well as concerns to supply shortage and quarantine, there was an overbuying of groceries and other daily necessities. This in fact resulted in a shortage of such products. If people had consumed products as usual, such a shortage would not have happened. This event implies that selfish behaviours are likely to cause an overall social loss as a consequence. These behaviours can be seen as a competition for resources.

1.2 Objective

In order to model such competition for resources as a non-cooperative game, a game of ”non-decreasing congestion games with load-dependent failures” defined by [8] is used throughout this project. To give a brief explanation of this model, the player’s utility is determined by cost and the number of resources he or she has chosen and the cost is determined by the number of other players who uses the same resource at the same time. The ”resources” in this model are not limited to specific items. It can be some facilities. For example, in the case scenario under the pandemic of COVID-19, using a set of crowded facilities involves a certain amount of cost that would reduce the player’s utility because of potentially less availability of items or services there. The detail of the model is explained in later chapters.

The main purpose of this project is to empirically evaluate the quality of a Nash equilibrium in the game obtained by running the algorithm designed by [8], and estimate the bounds on the price of anarchy and the price of stability. In order to obtain the data to evaluate, the developed software has two main functionalities: a functionality to find a Nash equilibrium in a game and a functionality to find the optimal solution in the same game.

1.3 Report Structure

This report starts with a review of existing studies regarding key theories relevant to this project: game theory, Nash equilibrium, congestion games, the price of anarchy and the price of stability. Chapter 3 and 4 explain the developed software from a high level to a low level. Chapters 4 also encompasses testing conducted for the software. After this chapter, professional issues and the code of conduct issued by the British Computer Society are discussed. Chapter 6 is very important; it provides data obtained from the system as an experiment and gives an evaluation of that by demonstrating graphs. Finally, this report ends with a conclusion consisting of a summary of the entire work and possible future work.

Chapter 2

Background

2.1 Literature Review

2.1.1 Game Theory

Game theory is the study of analysing strategies among multiple players. Its application is diverse ranging from social science to natural science. It has become common to see its intersection with the field of computer science in recent years [13]. Such an intersection has been established as an area called the algorithmic game theory that is aimed to design and analyse algorithms modelling strategies in games [10].

The following table 2.1 illustrates the famous example of a game, prisoner's dilemma cited from [1]. In the example, there are two players "A" and "B", two strategies "Cooperate" and "Defect". Each pair of values represents player A's utility on the left-hand side and player B's utility on the right-hand side. The higher utility to a player offers more benefits to the player. Each cell in which utilities are put is represented as a strategy profile.

A \ B	Cooperate	Defect
Cooperate	(3, 3)	(0, 5)
Defect	(5, 0)	(1, 1)

Table 2.1: Prisoner's dilemma [1]

2.1.2 Pure Strategy Nash Equilibrium

In the field of game theory, there is a concept of solution that is a description in which a game with particular characteristics may exhibit a certain result [7]. One example of such a solution is a dominant strategy equilibrium where there is a strategy profile in which every player chooses a strategy that maximises their utility without considering other players' strategy [7].

Contemporary, Nash equilibrium is the solution that is the most broadly used [7]. In contrast to a dominant strategy equilibrium, a strategy profile is said to be a Nash equilibrium if each player in the profile chooses a strategy that returns the highest utility taking account of other players' strategy [7]. Therefore, each player's optimal response meets the same strategy profile. Furthermore, when the choice of strategy is not stochastic, the strategy is said to be a pure strategy, and, given a set of strategy profiles, Nash equilibrium is said to be pure strategy Nash equilibrium when players are playing pure strategies. In the above table 2.1, the strategy profile that satisfies pure strategy Nash equilibrium is where both players choose the strategy "Defect" with the utility of 1.

2.1.3 Congestion Games

Congestion games are a class of games in the field of algorithmic game theory suggested by [9]. [9] replaced players' strategies as subsets of primary factors and introduced a notion of cost imposed on players by selecting a particular strategy. Cost is associated with each primary factor and its value is determined by the number of players using the associated primary factor. Since each strategy is a subset of primary factors, the total cost to the player is the sum of costs associated with the primary factor. It is proven that there is at least one pure strategy Nash equilibrium in such a class of games [9]. One example where this class is applied is a network of roads in which n people concurrently travel from one place to another [9]. The time to travel between them through any road is determined by the number of people using the road; more time is required as more people choose the same route. Each person tries to minimise the travel time by considering a trade-off between choosing the shortest paths and congestion. Since [9] the congestion games, various form of it has been modelled. For example, [4] introduced congestion games with player-specific payoff functions where players' payoff is only determined by the number of players employing the same strategy.

The model of game to be used in this project is congestion games with load-dependent failures [8]. What makes this model distinctive from the traditional congestion games is that it incorporates resource failure. Resource failure in this model means a resource may fail to work with a certain probability which is calculated by a function of the number of players who selected the same resource. The important point here is that, although the name of the model contains "congestion games", this class of games is completely different from the congestion games reviewed so far since it is not guaranteed to possess a Nash equilibrium. [8] identify that the congestion games defined by themselves, especially with decreasing cost functions, do not allow a potential function except for a special case. The potential function is a notion defined by [5] that changes its value when there is a change in utilities of a single player as a result of his or her unilateral deviation from a given strategy profile. These authors have proved that a potential game, a game possessing a potential function, always possesses a Nash equilibrium, and a congestion game is a subclass of potential games. Therefore, games without an admitted potential function are not always guaranteed to possess a pure strategy Nash equilibrium. However, [8] have found that there always exists a Nash equilibrium profile in congestion games with load-dependent failures under the situation that the cost is not decreasing, and designed a polynomial time algorithm to construct a pure strategy Nash equilibrium in such a game. The phrase ”not decreasing” means that the cost is either constant or increasing with congestion. Throughout this report, the abbreviation CGLFs is used to refer to ”non-decreasing” congestion games with load-dependent failures.

2.1.4 Price of Anarchy/Stability

The analysis is one perspective of algorithmic game theory. While a Nash equilibrium reflects the situation in which all players choose the optimal strategy given other players’ strategies, this is not always the strategy profile that reflects the optimal social utility, the sum of all players’ utility in a strategy profile that is highest of all the profiles. For example, in the table 2.1, the social utility of the Nash equilibrium profile is 2 while the optimal social utility in the game is 6. This implies that, if each player selfishly seeks higher profit from each other, it potentially ends up making an overall loss to each other compared with the optimal solution.

The concept of the price of anarchy is one of the tools used to analyse the quality of equilibrium compared to the optimal solution[2]. This concept was first introduced by [3] by questioning how bad the ratio between the optimal solution and the worst Nash equilibrium in a game where players share common resources and act selfishly can be.

In a utility maximisation game, the price of anarchy is calculated by dividing the maximum social utility, which is not limited to be in equilibrium, by the worst social utility which is in equilibrium.

$$PoA = \frac{\max_{s \in S} Welf(s)}{\min_{s \in Equil} Welf(s)} \tag{2.1}$$

By contrast, in a cost minimisation game, it is calculated by dividing the maximum social cost of an equilibrium profile by the minimum social cost[11]. These imply that as the value of the price of anarchy approaches 1, it reduces the inefficiency of the quality of the equilibrium.

$$PoA = \frac{\max_{s \in Equil} Cost(s)}{\min_{s \in S} Cost(s)} \tag{2.2}$$

However, if a game contains multiple equilibria and one of them is highly inefficient, the value of the price of anarchy easily becomes large no matter how the other equilibria are efficient. In order to make such games distinctive from games with which all equilibria are inefficient, the notion of the price of stability is introduced [12]. The price of stability measures the inefficiency of the best equilibrium. Analogous equations are derived.

$$PoS = \frac{\max_{s \in S} Welf(s)}{\max_{s \in Equil} Welf(s)} \tag{2.3}$$

$$PoS = \frac{\min_{s \in Equil} Cost(s)}{\min_{s \in S} Cost(s)} \tag{2.4}$$

Because of its definition, the price of stability is situated between 1 and the value of the price of anarchy in a game. Furthermore, it is possible that the price of anarchy and the price of stability are identical if a game possesses only one equilibrium. The following equality and inequality relation can be derived.

$$1 \leq PoS \leq PoA \tag{2.5}$$

Given the equations 2.1 and 2.3, in the table 2.1, the optimal solution is the profile in which both players choose the strategy "Cooperate" with the value of 6. The social utility of the Nash equilibrium profile, in which both players choose the strategy "Defect", is 2. Since the game contains only one Nash equilibrium, the value of both of price of anarchy and price of stability is 3 implying the equilibrium is 3 times as inefficient as the optimal solution.

There are numerous studies to measure how inefficient the equilibrium of a certain congestion game can be by finding the price of anarchy and the price of stability of it. For example, [2] theoretically shows that the value of the price of anarchy of congestion games with linear latency functions depends on the number of players when the game is asymmetric and the social cost is maximum. On the other hand, neither theoretical nor empirical evaluation of the Nash equilibrium computed by the algorithm designed by [8] has been conducted yet. This is why this project undertakes an empirical evaluation of the ratio between the obtained equilibrium and the optimal solution.

Chapter 3

Specification & Requirement

The purpose of this project is to empirically evaluate the quality of Nash equilibrium found in a CGLF by using the algorithm in [8]. In order to obtain a set of the value of the ratio between the equilibrium and the optimal solution for empirical analysis, software that builds a game environment defined in [8] and finds pure strategy Nash equilibrium profile has been developed. This chapter first describes the specification of the software by referring to [8]. It then lists the requirements to be achieved. Explanation in this order would encourage understanding of the software to be produced.

3.1 Specification

3.1.1 Definition of the Games

The following are the elements of CGLFs [8]. N represents a set of players, M represents a set of resources, f represents a failure probability function, whose output is between 0 and 1, given congestion as an integer value and c represents a cost function, whose output is a non-negative real number, given congestion as an integer.

$$N = \{1, 2, ..., n\}$$

$$M = \{e_1, e_2, ..., e_m\}$$

$$f : \{1, 2, ..., n\} \to [0, 1)$$

$$c : \{1, 2, ..., n\} \to R^+$$

At game construction, given a set of players and a set of resources, it constructs strategy profiles of all combinations of strategies and players. While players are different from each other, resources are identical to each other. An element of strategies, i.e. a strategy, is a subset of the resources, and the strategy set is a power set of resources. The obtained strategy profiles are later compared with themselves to find the optimal solution (a strategy profile with the maximum social utility) in the game.

The utility U of each player is calculated using the following function [8]. Note that the player's utility is 0 when the player does not choose any resources.

$$U_i(\sigma) = (1 - \prod_{e \in \sigma_i} f(h_e(\sigma)))v_i - \sum_{e \in \sigma_i} c(h_e(\sigma)) \tag{3.1}$$

- i: player's index
- σ: strategy profile
- σ_i: player i's strategy
- v_i: player i's benefit
- $h_e(\sigma)$: congestion of resource e under a strategy profile σ

The following table 3.1 is an example of a strategy profile. Assume there are players 1 and 2, and resources e_1 and e_2. Players 1 and 2 have the benefits 4 and 2, the cost for the resource with congestion 1 and 2 is 1 and 3, failure probability for the resource with congestion 1 and 2 is 0.01 and 0.5, respectively. The number on each cell is the player's index. It shows that the resource

e_1	e_2
1	1
	2

Table 3.1: An example of a strategy profile

e_1 is used by only player 1; therefore the cost for the resource e_1 is $c(h_{e1}(\sigma)) = c(1) = 2$ and the failure probability is $f(h_{e_1}(\sigma)) = f(1) = 0.01$. The resource e_2 is used by both player 1 and 2; therefore the cost for the resource e_2 is $c(h_{e_2}(\sigma)) = c(2) = 3$ and the failure probability is $f(h_{e_2}(\sigma)) = f(2) = 0.5$. The utility for the player 1 is $U_1(\sigma) = (1 - 0.01) \times 4 - (3) = 0.96$, and the utility for the player 2 is $U_2(\sigma) = (1 - 0.01 * 0.5) \times 2 - (1 + 3) = -2.01$. The social utility, as mentioned before, is the sum of all player's utilities. Therefore, in the example strategy profile above, the social utility is $U_1(\sigma) + U_2(\sigma) = -1.05$.

3.1.2 Finding a Nash equilibrium

Nash equilibrium can be found by performing a series of single operations to ”post-addition D-stable strategy profile” [8]. A series of operations refers to ”D-move” denoting dropping a resource of a player under strategy profile, ”A-move” denoting adding a resource to a player and ”S-move” denoting switching player’s resource with another resource, and ”post-addition D-stable strategy profile” refers to a strategy profile σ in which dropping a resource added by performing A-move to σ is not profitable. The operations are conducted when they are profitable. That is, the difference between the utility after performing an operation and the original utility is positive. For example, the following inequality must be satisfied to find D-move with a resource a profitable.

$$\begin{aligned} U_i(\sigma_i \setminus \{a\}, \sigma_{-i}) &> U_i(\sigma) \\ (1 - \prod_{e \in \sigma_i \setminus \{a\}} f(h_e(\sigma)))v_i - \sum_{e \in \sigma_i \setminus \{a\}} c(h_e(\sigma)) &> (1 - \prod_{e \in \sigma_i} f(h_e(\sigma)))v_i - \sum_{e \in \sigma_i} c(h_e(\sigma)) \\ c(h_a(\sigma)) &> v_i(1 - f(h_a(\sigma))) \prod_{e \in \sigma_i \setminus \{a\}} f(h_e(\sigma)) \end{aligned}$$

Similarly for A-move with a resource a,

$$U_i(\sigma_i \cup \{a\}, \sigma_{-i}) > U_i(\sigma)$$

and S-move with a resource a and a resource b that player i did not have

$$U_i(\sigma_i \cup \{b\} \setminus \{a\}, \sigma_{-i}) > U_i(\sigma)$$

The state where each operation is no longer necessary is denoted as D-, A- and S-stable. When these operations are no longer necessary, i.e. a strategy profile is D-, A- and S-stable at the same time, the profile is a Nash equilibrium. Therefore, in order to obtain a Nash equilibrium, the system must perform these operations and then reach the stabilities. How these operations are performed is illustrated in the next chapter.

In order to execute the equilibrium finder correctly, the following constraints must be followed.

1. The number of players must be more than 1.

2. The number of resources must be more than 1.

3. The cost function of a resource must be non-negative and non-decreasing.

3.1.3 Measuring the Ratio

Each player in the CGLFs tries to maximise their expected utility [8]. Therefore, the optimal strategy profile is one in which the sum of all players' utilities (a social utility) is the highest of all strategy profiles. The ratio is then calculated by dividing the social utility of the optimal strategy profile by the social utility of an obtained Nash equilibrium.

3.1.4 Limitation

Since the software is developed as a tool to obtain a set of the value of the ratio, there is no functionality for users to interact with it via the graphical user interface. In order to obtain different data with different inputs, potential users are expected to run the software via the command-line interface as well as manually changing inputs on a code editor. When executing the system, potential users are expected to follow all the theoretical constraints defined in [8] to ensure the equilibrium finder correctly works, for example, the number of players cannot be 1.

The detail on how to obtain data using the software is discussed in Chapter 6.

3.2 Requirement

As mentioned above, the software developed for this project has two main functionalities: finding a Nash equilibrium in a CGLF [8] and finding the optimal solution in the game. In order to develop such software, the following requirements are elicited. In this section, requirements are listed in two subsections: one for the CGLF and the other for the algorithm to find a Nash equilibrium. Although the actors in the requirement for equilibrium finder are the same as the requirements for the CGLF environment, they demand additional requirements. Unless specified, the actors in the requirement of equilibrium finder take over the requirements for the CGLF environment.

3.2.1 CGLF Environment

Player Requirements

1. The player must choose a subset of resources at the same time as the other players choose.

2. The player must maximise his expected utility.
3. The player must possibly share the same resources as other players.
4. The player must have the benefit from successful task completion.

Resource Requirements

1. The resource must have cost.
    - The cost must be a non-negative real number.
    - The cost must not decrease as more players use the resource.
2. The resource must have failure probability.
    - The failure probability must be in the range $0 \leq probability < 1$.
    - The failure probability must not decrease as more players use the resource.
3. The resource must be identical to the other resources apart from the congestion on itself.

System Requirements

1. The system must build strategy profiles given players and resources.
2. The system must find the optimal strategy profile.
3. The utility for the player must be calculated given a strategy profile.
4. The strategy must consist of a subset of resources.
5. The strategy set must be a power set of resources.
6. The congestion on resources must be from 1 to the number of players.
7. The number of players must be more than 1.
8. The number of resources must be more than 1.

3.2.2 Equilibrium finder

Unless specified, all the requirements that are already assigned to the actors are transferred from above.

Player Requirements

1. The player must add one resource at one time.

2. The player must drop one resource at one time.

3. The player must switch one resource with another resource at one time.

System Requirements

1. The system must find a Nash equilibrium by appropriately performing A-, D- and S-moves.

The following figure is a class diagram representing the system for the CGLF construction and finding a Nash equilibrium. Each of Strategy Profile, CGLF and Equilibrium Finder takes two or more than two players and resources. Since the number of players and resources is both more than one, and therefore, each player has at least 4 strategies, CGLF builds at least 16 strategy profiles. The member variables in the Equilibrium Finder, such as k, xD, are explained in the next chapter.

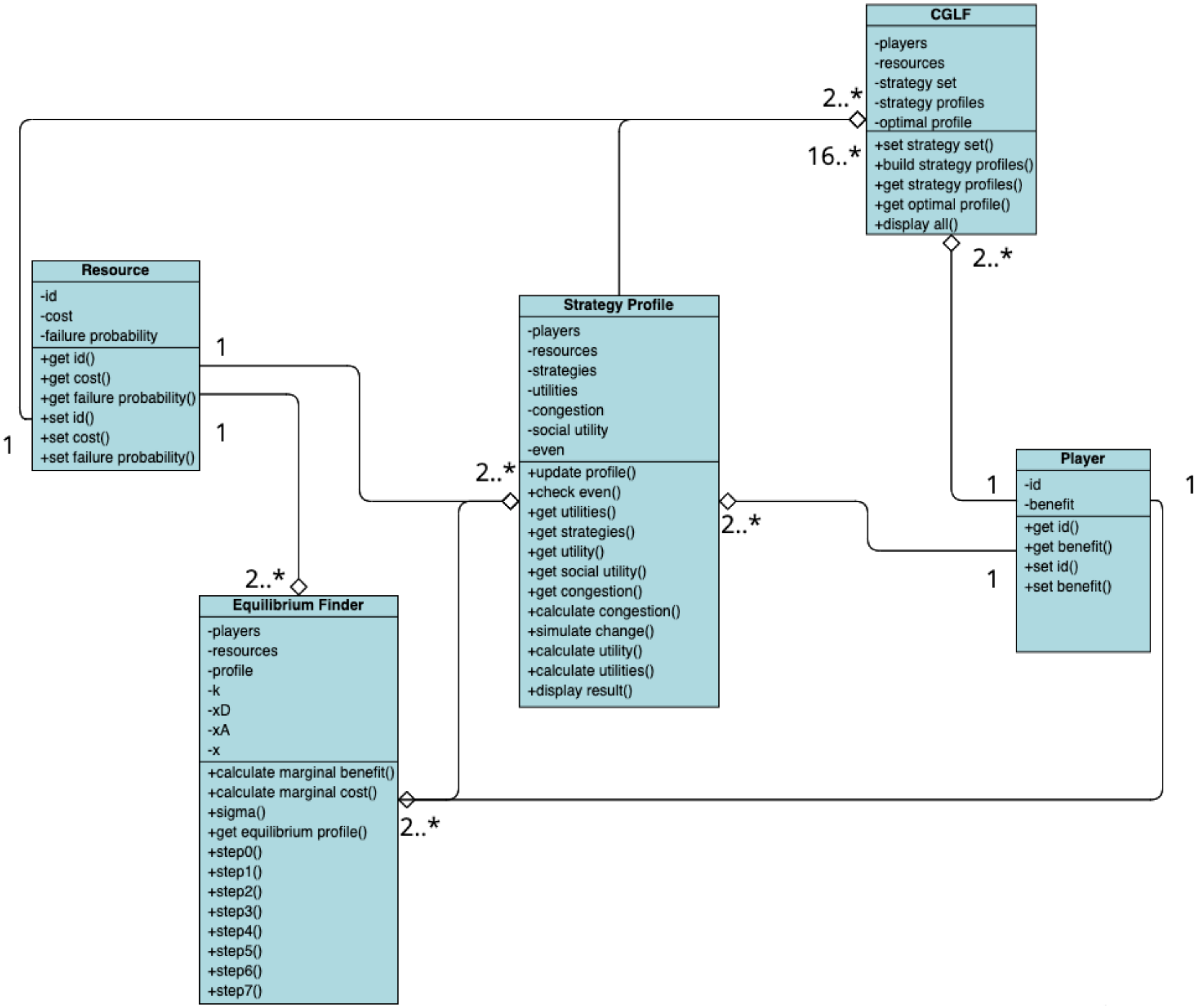

Figure 3.1: A class diagram of the system

Chapter 4

Design & Implementation

The software developed through this project is divided into two major functionalities: finding a Nash equilibrium and finding the optimal solution. It was developed using Python language because of its simplicity of syntax and operation confirmation.

4.1 Preparation

Throughout developing software, the dictionary data structure is used for the majority of collections, and the list data structure is rarely used. This is because there are considerable situations where the index is very important to retrieve data. For example, when calculating the cost and failure probability function of a resource, these value depends on the number of players who is using the resource, i.e. the input to this function is a natural integer. At first glance, these functions can be coded with a list data structure. However, the list data structure's index always starts from 0. Although this issue could be addressed by assigning an index minus 1, this method is likely to cause confusion during the development of the software. Therefore, in order to avoid this drawback, the dictionary data structure, where any data can be assigned as a key, is employed.

When inserting an item of the game object into a collection, such as a player and a resource, their index is inserted rather than the object itself. This is because of convenience. During the implementation of this software, an object of customised classes, such as Player and Resource, is difficult to check its identity when errors occur since printing these objects displays their memory location of them. However, by inserting indices instead, it displays primitive data and facilitates resolving errors.

The key objects when modelling a CGLF [8] are players and resources. They are implemented as a class. Class Player contains the player's index and his or her benefit from successful task completion as attributes. Class Resource contains resource's index, cost function and failure probability function as attributes. The cost function and the failure probability function are implemented with a dictionary data structure. Their key is congestion, which is a positive integer, and once the key is passed, it returns cost and failure probability, respectively. Note that all resource instances have the same cost function and failure probability function, and the only difference between them is their indices.

When finding an equilibrium profile of a CGLF with the resources and players, [8] specifies the constraints that the cost function and the failure probability function are non-decreasing, and player's benefits are assigned in descending order. However, these constraints are not implemented inside these classes. This is because each value of the parameters is dependent on what scenario a user of this software is simulating; therefore, a certain non-decreasing function for each parameter (and non-increasing for the player's benefit) must be designed by the user. These classes are used to implement all of the following classes: StrategyProfile, CGLF and Equilibrium.

4.2 Building CGLF

4.2.1 Class StrategyProfile

The key role of this class is to store the utilities of players under the combination of strategies and congestion on each resource. The following is a python code showing a constructor of the class StrategyProfile.

```python
class StrategyProfile():
    def __init__(self, strategies: Dict[int, Set[int]],
                       players: Dict[int, Player],
                       resources: Dict[int, Resource]):
        self.__strategies: Dict[int, Set[int]] = strategies
        self.__players: Dict[int, Player] = players
        self.__resources: Dict[int, Resource] = resources
        self.__utilities: Dict[int, float] = dict()
        self.__congestion: Dict[int, int] = dict()
        self.__even: int = None
```

```
self.__social_utility: float = None
self.update_profile()
```

The constructor takes three parameters as an input: *strategies*, *players* and *resources*. *strategies* is a dictionary data structure whose key is a player's index and value is a strategy that the player chooses. The strategy is a set of resources indices. *players* is a dictionary data structure whose key is a player's index and value is a player instance. *resources* is a dictionary data structure whose key is a resource's index and value is a resource instance. With this input, utilities of each player and congestion on each resource are calculated using the formula 3.1.

In addition to these, the class contains the functionality of simulating whether changing a certain player's strategy is beneficial. This function later contributes to finding a Nash equilibrium. Although this is a simple function, the implementation was carefully conducted by using a *deepcopy*, which creates a new object and then inserts the value of the original object into it, in order to protect the original instance variable, self.strategies. Since this instance variable contains Set as illustrated in the code, which is an immutable object, changing the value in any object copied apart from *deepcopy* affects the original variable. The last functionality which is worth discussing is updating a profile. This function makes important variables, such as congestion and utilities, up-to-date. This function is also simple but important when the profile is manipulated outside. For example, when finding Nash equilibrium, there is a case where a further resource is assigned to a certain player. This impacts congestion and utilities in a profile. Without updating the profile's state, it results in returning a wrong equilibrium.

4.2.2 Class CGLF

The instance of this class is used to find the optimal strategy profile. In order to conduct this, the class creates all strategy profiles given a set of players and a set of resources as an input, and then finds the optimal one. The constructor takes two parameters as input in the same way as the Class StrategyProfile does: *players* and *resources*. The strategy set for the players is obtained by creating a power set of the set of resources. Players' strategy is an element of this set. The process of creating all strategy profiles given the input is done by generating a Cartesian product of sets of the strategy set each of which is indexed by the player's index. Since this class is used only to obtain the optimal solution given the input of players and resources, the creation of strategy profiles can be completed inside its constructor by calling relevant functions.

4.3 Building Nash Equilibrium Finder

In order to implement the algorithm to find a Nash equilibrium, a python file containing a single class for the algorithm is built. This class interacts with the StrategyProfile class but does not have access to CGLF class. As mentioned before, the condition for a strategy profile to be Nash equilibrium is to achieve D-stability, A-stability and S-stability at the same time. When the algorithm is executed, the common congestion k is the same as the number of players. This is the situation where all the players hold all the resources. Such a profile is called k-even strategy profile. For example, the following table 4.1 is modified from table 3.1. This table illustrates the profile where $k = n = 2$ since all the players are holding all the resources. The terms

e_1	e_2
1	1
2	2

Table 4.1: An example of a strategy profile

σ-light and σ-heavy are used to refer to a certain resource. A σ-light resource is a resource with the minimum congestion in a strategy profile while a σ-heavy resource is any resource except for σ-light ones. It is possible that there are several σ-light resources at the same time. For example, the σ-light resource in the table 3.1 is e_1 while all the resources in the table 4.1 are σ-light. The following describes the process of the algorithm designed by [8].

- **Step 0: Check D-stability at n-even profile**

  This step checks whether holding all the resources is profitable to all the players. In other words, this is ensuring no player gets benefits from dropping any of the resources the player is holding. Since all the players have all the resources, A- and S-stability is satisfied. The important notice here is that ensuring the D-stability of player n, who has the least benefit value and ordered lastly, ensures all the other players' D-stability at the same time. If the condition is satisfied, which implies D-stability is satisfied, the algorithm terminates by returning a strategy profile in which all the players hold all the resources. If this is not satisfied, it decrements the value of k by 1, then proceeds to Step 1.

- **Step 1: Determine the value of k**

  This step determines the value of k by finding the maximum number of resources that each player keeps without dropping in a k-even strategy profile. Let $x^i_D(k)$ denotes such number where i denotes player i. If the sum of $x^i_D(k)$ for all the players is less than the number of the entire resources multiplied by the value k, it decrements the k by 1, then proceeds to Step 2; otherwise proceeds to Step 3.

- **Step 2: Construct an equilibrium profile if $k = 0$**

  In this step, if the value of k is equal to 0, which is the situation where no resource is assigned to all players and D- and S-stability is satisfied, each player is assigned $x^i_D(1)$ resources. Note that the resource with the least congestion is allocated first. The algorithm terminates by returning an equilibrium profile since A-stability is satisfied by the resource allocation. If the k is more than 0, it goes back to Step 1.

- **Step 3: Check the existence of a k-even equilibrium**

  This step checks the existence of a k-even equilibrium by finding the minimum number of resources with which each player is no longer required to conduct an A-move in a k-even strategy profile. Let $x^i_A(k)$ denotes such number where i denotes player i. If the sum of $x^i_A(k)$ for all the players is more than the number of the entire resources multiplied by the value k or if there is any player i whose $x^i_A(k)$ is more than $x^i_D(k)$, it proceeds to Step 5; otherwise proceeds to Step 4.

- **Step 4: Construct a k-even equilibrium profile**

  In this step, a k-even equilibrium profile is constructed by assigning for all players with the index ($i = 1$ to n) at least $x^i_A(k)$ resources and at most $x^i_D(1)$ resources. Every resource is assigned to exactly k players in the manner that the resource with the least congestion is allocated first. In order to achieve this, a formula d_i is used. d_i represents the number of resources that player i can use in addition to $x^i_A(k)$. $\sum(\sigma)$ denotes the number of resources that are already assigned and $\sum x^i_A(k)$ denotes the number of resources that must be saved for the later players to obtain resources. Once the resource allocation completes, the algorithm terminates by returning the k-even equilibrium profile.

- **Step 5: Construct a post-addition D-stable profile**

  This step constructs a post-addition D-stable profile. Firstly, it finds the maximum number of resources that each player keeps without dropping in a k-even post-addition D-stable profile. Let $x^i(k)$ denotes such number where i denotes player i. Next, as long as the number of resources assigned to players so far does not exceed the number of the entire resources multiplied by k, each player is assigned resources, and the number is the minimum of either $x^i(k)$ or the rest of resources. The constructed profile is passed to Step 6.

- **Step 6: Check A-stability**

  Given a post-addition D-stable profile, this step checks whether the profile is A-stable. This process is conducted by finding a resource with minimum congestion a for all players $i \in N$ that was not previously assigned to i. If there is any player i who gets benefit from adding such a resource, the algorithm proceeds to Step 7 by passing the profile and the data of players and resources that are to be used for the A-move. If not, the A-stability is satisfied; therefore, the algorithm terminates by returning the passed strategy profile.

- **Step 7: Conduct a one- or two-step addition**

  Given a profile and players requiring an A-move with a certain resource, the algorithm conducts an A-move to player i requiring a resource a^* with minimum congestion among resources to be used for A-move. If a^* is σ-light, it is allocated to the player i. If not, the algorithm finds a σ-light resource b and a player j who has a but not b. Once such a player and resource are found, player j switches from resource a to b, and player i obtains a resource a. After the allocation, the algorithm goes back to Step 6.

Since the purpose of this class is only to find a Nash equilibrium, it runs Step 0 inside the constructor, and store the obtained equilibrium profile as an instance variable. As demonstrated above, each step is connected to each other; therefore, running Step 0 explores all the other steps if required.

4.4 Software Testing

In order to ensure the software works correctly, unit testing was conducted. The testing in this project covers the software of the CGLF constructor and the Nash equilibrium finder. Testing for the software that generates data (a set of values of the ratio) was not conducted since the success of the testing for the CGLF constructor and the equilibrium finder ensures the output of data.

4.4.1 Unit Testing

Unit testing was conducted for major classes: StrategyProfile, CGLF and Equilibrium. Testing for the other classes, i.e. Player and Resource, was not conducted since they consist of only simple functions of accessing and mutating their member variables.

The key functionality of CGLF class is to create a strategy set, which is a power set of resources, and strategy profiles. Testing the strategy set creation is done by comparing the result of the actual function with an expected set which is manually inputted. On the other hand, since the class StrategyProfile is a reference type, it is unable to compare the equality of the created instances with those manually created with the same input. In order to overcome this issue, the value of utility that the class has as an instance variable is used. By adding these values of utilities in a set, it passes the unit test. However, this only confirms the equality of social utilities of both sets of strategy profiles. It is not negligible that the social utilities coincidentally match and other instance variables do not match.

It cannot be said that the testing for the functionality of finding a Nash equilibrium is comprehensively conducted. The testing measure for it in this project was to compare the social utility of returned Nash equilibrium profile with the expected social utility of Nash equilibrium that was calculated by hand. The major obstacle is that it is difficult to check whether the output of the algorithm is Nash equilibrium profile as the number of players or resources increases. In order to conduct unit testing for the algorithm, a game in the following table is considered.

A \ B	ϕ	e_1	e_2	e_1, e_2
ϕ	(0.000, 0.000)	(0.000, 2.960)	(0.000, 2.960)	(0.000, 2.000)
e_1	(0.089, 0.000)	(-1.186, 0.960)	(0.089, 2.960)	(-1.186, 0.990)
e_2	(0.089, 0.000)	(0.089, 2.960)	(-1.186, 0.960)	(-1.186, 0.990)
e_1, e_2	(-0.900, 0.000)	(-1.903, 0.960)	(-1.903, 0.960)	(-2.974, -0.270)

Table 4.2: CGLF utility table for unit testing

The input to create this game is:

1. the number of players: 2 (A and B)
2. the number of resources: 2 (e_1 and e_2)
3. A's benefit: 1.1
4. B's benefit: 4
5. cost: $C(1) = 1$, $C(2) = 2$
6. failure probability: $P(1) = 0.01$, $P(2) = 0.26$

The value on the left-hand side of each profile represents the utility of player A and the value on the right-hand side represents the utility of player B. The strategies on the left column represent player A's strategies and the strategies on the top column represent player B's strategies. In this game, the Nash equilibrium profiles are the profile in which player A chooses the strategy e_1 and player B chooses the strategy e_2 and player A chooses the strategy e_2 and player B chooses the strategy e_1. Note that, however, the algorithm returns only one pure strategy Nash equilibrium profile. If testing for this algorithm is conducted after manually finding a Nash equilibrium, such as by drawing a table as demonstrated, it immediately reaches a limit. The number of profiles exponentially increases by m^n where m represents the number of resources and n represents the number of players. Furthermore, two-player games can be drawn on one table, while it is unable to draw games with more than two players on one table.

Chapter 5

Legal, Social, Ethical and Professional Issues

5.1 Ethical Concerns

As mentioned at the background chapter, game theory is involved in a wide range of fields as a way to make decisions not limited to the field of computer science but also social sciences, such as politics and economics, ant etc. The decisions made by these fields potentially lead to some ethical concerns.

In the case of CGLFs, an obtained Nash equilibrium is sometimes inefficient compared with the optimal solution. This implies that selfish behaviours can degrade social utility. Therefore, following a strategy of a Nash equilibrium, whose quality is inefficient, may not be ideal in certain scenarios such as ones involving public well-being. Imagine now the society under the COVID-19 pandemic is modelled by using a CGLF, where players are citizens and resources are facilities. The authority is taking countermeasures against the pandemic, one of which is to "ask" citizens not to use facilities in order to prevent the potential clusters and outbreaks while citizens wish to use them. Let the optimal solution here be the situation where all players do not use any facilities. If citizens do not follow the countermeasure, the pandemic may become worse. It is possible that players use facilities potentially because of low cost or low failure probability. Note that cost here is not limited to money. In this scenario, a possible Nash equilibrium for players might be to use some resources. However, this could be inefficient for society and prolong the pandemic. In other words, following a Nash equilibrium of this scenario may degrade public well-being. Therefore, potential users of the system developed through this project must be careful whether using the system in a certain scenario and following a strategy of a Nash equilibrium is ethically right.

5.2 British Computer Society Code of Conduct and Code of Good Practice

Throughout this project, the rules issued by British Computer Society Code of Conduct and Code of Good Practice were followed. The developed system does not contain malicious functionalities that commit immoral activities. However, as mentioned in the previous section, users of the software must consider the impact of following a strategy of a computed Nash equilibrium to the public well-being if scenarios such as social phenomena are modelled. Additionally, the data to be generated from the system, which is to be explored in the next chapter, does not violate the rules mentioned above and is obtained in the unbiased manner.

Chapter 6

Results/Evaluation

6.1 Limitations

As mentioned in the previous chapter, the ratio of social utilities between optimal solution and equilibrium solution is calculated using the equation 2.1. This equation is used since players in the game[8] try to maximise their utility. The value of social utility for the equilibrium point is pure strategy Nash equilibrium.

In order to observe the change in the ratio, five parameters are manipulated: the number of players, the number of resources, player's benefit, resource cost and resource failure probability. The following listings illustrate the detail of the parameters with the value range and functions to calculate the value of parameters that varies depending on the number of players.

1. number of players: $n = \{2, 3, 4\}$ (default $n = 4$)
2. number of resources: $m = \{2, 3, 4\}$ (default $m = 4$)
3. initial benefit: $v = \{70, 80, 90, ..., 170\}$ (default $v = 100$)
4. initial cost: $c = i * j$ j= $\{0, 1, ..., 10\}$ (default $j = 5$)
5. initial failure probability: $f = 1 - \frac{i}{1+\frac{j}{10}}$ j = {0,1,...,10} (default $j = 5$)
6. $i = 1, 2, 3, ..., n$

The range of value is decided as in the above because of the complexity of the software. Although the complexity of the algorithm of finding a Nash equilibrium [8] is $O(n^2m + nm^2)$ and computes the equilibrium in a game of more players and resources than the range in this project, the algorithm to build a CGLF is not robust enough. The maximum number of players that does not exceed the maximum run time is 10 when the number of resources is 2, and the maximum number of resources is 11 when the number of players is 2. In this experiment, the maximum number of players and resources that can be at the same time is 4. The upper bound of the failure probability is 0.8 with the resource congestion of 4, the upper bound of cost is 40 with the resource congestion of 4, and the lower bound of player's benefit is 54. The value of each bound can be calculated using equations in the following segment of code showing how the program assigns value to each parameter. Note that the player's benefit must be assigned decreasingly as in the code below, i.e. the first player is assigned the highest benefit and the last player is assigned the lowest benefit. Furthermore, the cost and the failure probability for each congestion must be assigned non-decreasingly since these are the constraints when the equilibrium finder works correctly.

```
for i in range(1, num_players + 1):
    players[i] = Player(i, benefit)
    benefit = initial_benefit - (i + 1) ** 2
failure_probability = dict()
for i in range(1, num_players + 1):
    failure_probability[i] = 1 - 1 / (1 + i / 10 * start_probability)
cost = dict()
for i in range(1, num_players + 1):
    cost[i] = i * (start_cost + 1)
```

Each function is formed not to violate the constraint in [8]. For example, with the above functions for the cost and failure probability of a resource, the value of cost and failure probability of a more congested resource is never less than those of a less congested resource. On the other hand, the value of the benefit for players is assigned decreasingly. This is because, when dealing with players, the algorithm finder assumes players with more benefit are ordered first.

The experiment was conducted to obtain the four different type of data set. For each experiment, two variables are changed and the other three are fixed at each default value. It would be valuable if the value of every variable was changed concurrently during the simulation. However, this approach was not employed since there is no method to visualise a 5-dimensional graph. The combinations of variables are: player and resource, benefit and cost, benefit and failure probability, and cost and failure probability.

6.2 Results

The following figures illustrate the results of the ratio obtained. The left axis, whose scale starts from 0 and ends between 1.5 and 2.5, shows the value of the ratio, and the other two axes show the value of variables. As mentioned in Chapter 2, the price of anarchy is the ratio between the value of social utility of the optimal solution and the value of social utility of the worst equilibrium solution. This implies that the value of the price of anarchy of a game in each experiment can be assumed to be at least the ratio obtained as a result and the value of the price of stability of the game can be assumed to be at most the ratio. The obtained equilibrium solutions are considered to be less inefficient when the ratio is close to 1, and the solution is more inefficient otherwise.

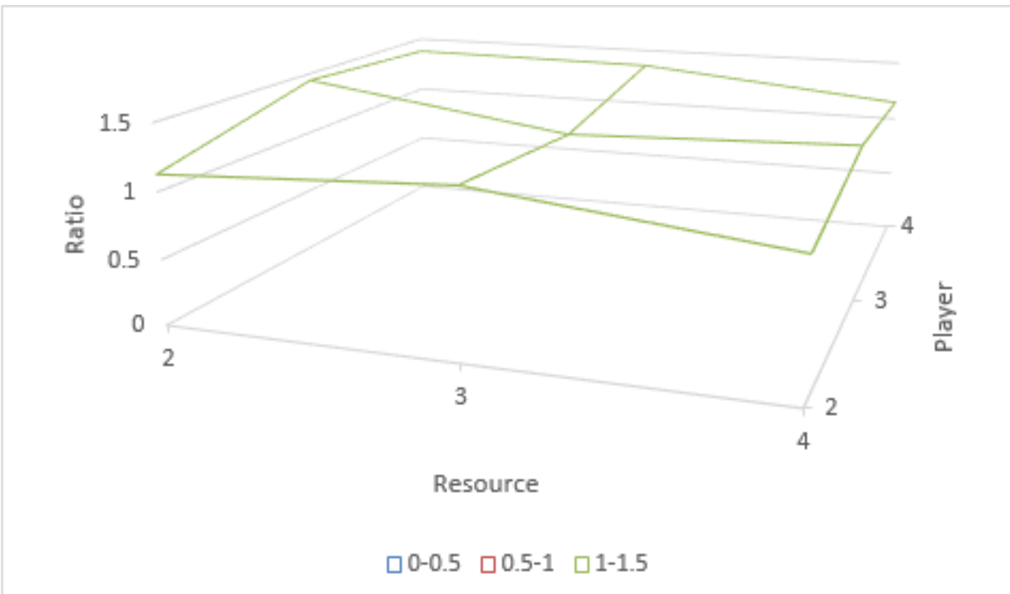

Figure 6.1: Setting: player's benefit = 100, cost = 5, failure probability = 0.33, max = 1.44

The figure 6.1 illustrates the results of the ratio by changing the number of players and resources. Apart from the case where the number of resources and players is 4 and 2, respectively, the graph shows a lower ratio than the others when the number of resources and players is the same as each other.

The figure 6.2 illustrates the results of the ratio by changing the value of initial failure probability and player's benefit. It is clear that the ratio is 1 when the initial failure probability is 0. Overall, the ratio is increasing as the value of the initial failure probability increases.

The figure 6.3 illustrates the results of the ratio by changing the value of the initial cost and player's benefit. It is clear that the ratio is 1 when the initial cost is 0. While any trend is not

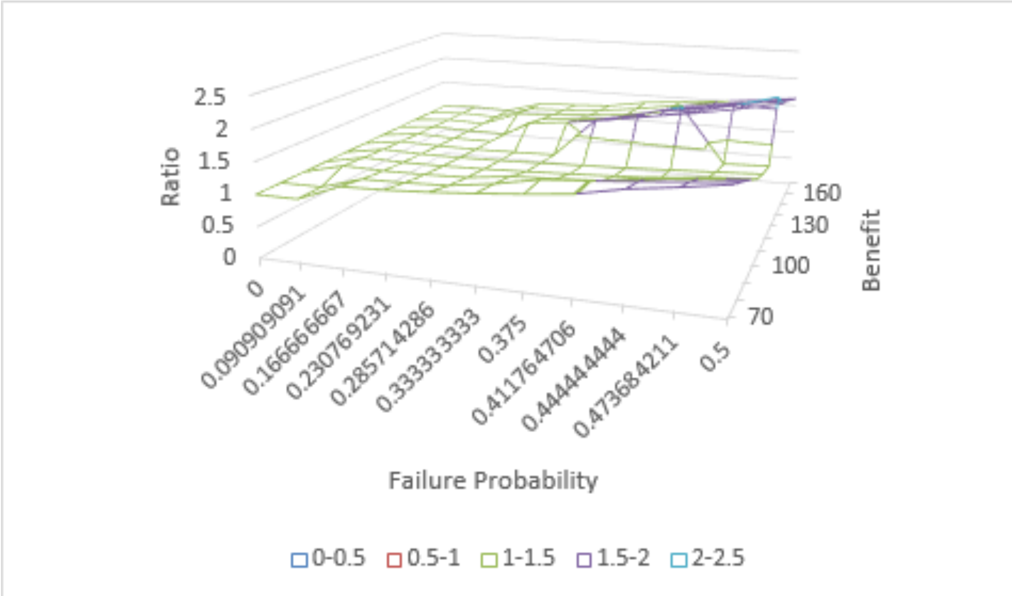

Figure 6.2: Setting: number of players = 4, number of resource = 4, cost = 5, max = 2.17

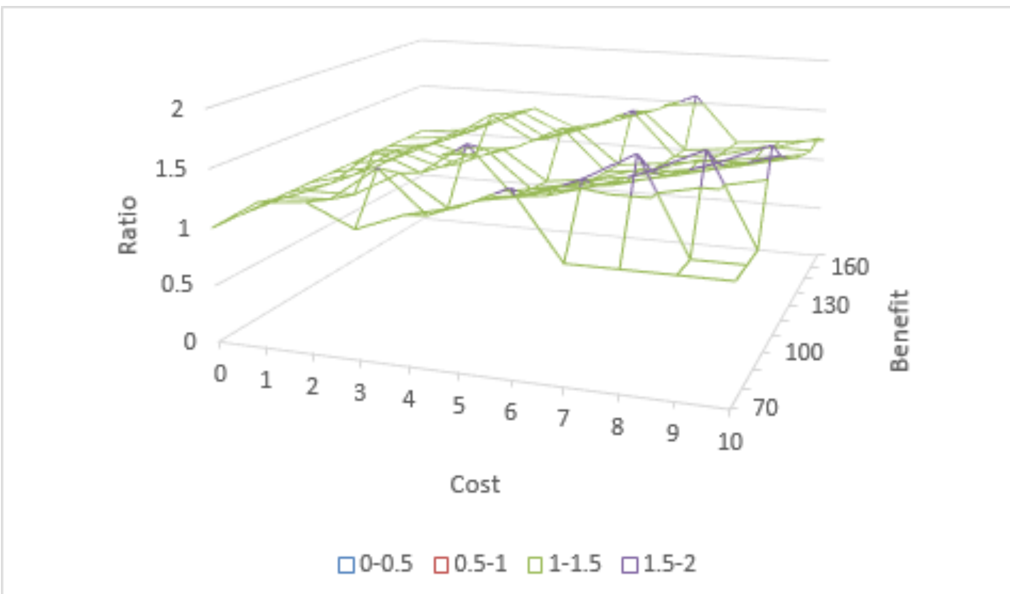

Figure 6.3: Setting: number of players = 4, number of resource = 4, failure probability = 0.33, max = 1.80

found in terms of the change in the value of the benefit, it is likely that the rise in the initial value of cost slightly increases the ratio. However, this observation is through monitoring the maximum value of the ratio for each cost value. Additionally, observing the area around the value setting of 10 for cost and 70 for benefit, the graph is showing the ratio of 1.

The figure 6.4 illustrates the results of the ratio by changing the value of initial cost and failure probability. As in the other figure containing failure probability as an axis, the ratio is

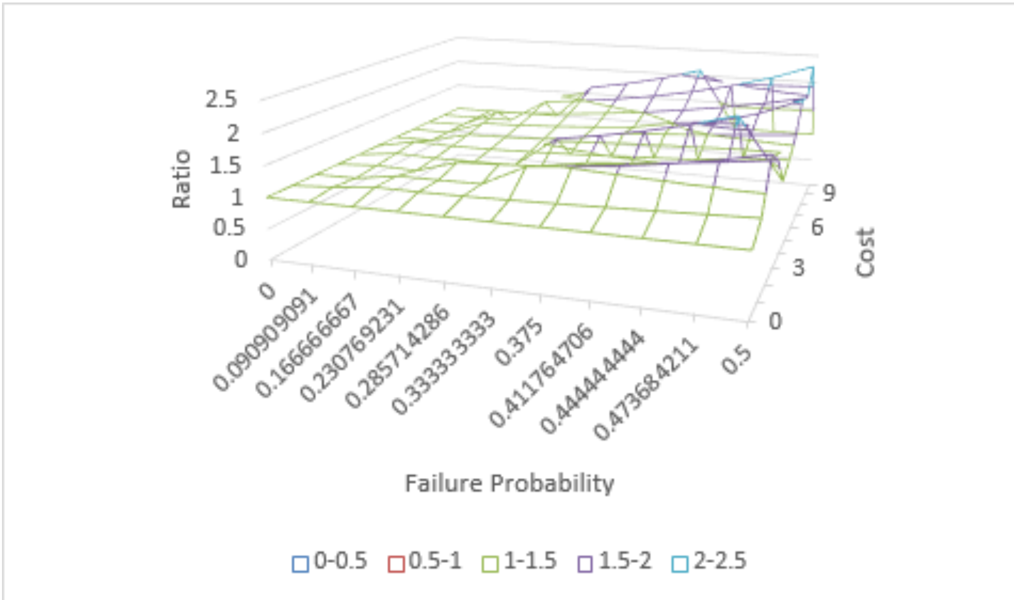

Figure 6.4: Setting: number of players = 4, number of resource = 4, player's benefit = 100, max = 2.41

always 1 when the failure probability is 0, and there is an increase in the ratio as the failure probability increases. Similarly, the ratio is always 1 when the failure probability is 0. Although this graph does not show the trend found in 6.3 that the ratio increases as the cost increases, this graph is showing the similar trend found in 6.3 that the ratio of 1 is gathered around the value setting of 10 for cost 0.09 for failure probability.

By observing the change in the ratio by the value of failure probabilities in 6.4 and 6.2, it is likely that the quality of a Nash equilibrium produced by the algorithm [8] is relatively affected by the value of failure probability. By observing the figures 6.4 and 6.3, it demonstrates that the ratio is 1 when the initial cost is 10 and the value of the other axis is low, such as the situation where the initial failure probability is 0.09.

The highest ratio in this experiment was 2.41. This situation occurred in the game in which the input cost was 9 and the initial failure probability was 0.5. This implies that, under certain parameter settings, the algorithm designed by [8] computes a Nash equilibrium that is 2.41 times as inefficient as the optimal solution. From this result, it can be derived that the upper bound of the price of anarchy of Nash equilibria that the algorithm computes is at least 2.41 and the upper bound of the price of stability is at most 2.41. The analysis conducted in this experiment might become invalid when one with a different scope is conducted. For example, the potential experiment that might lead to a different analysis is one manipulating the parameters of player and resource at the same time. In this experiment, the range of the number of players and resources was from 2 to 4. This range might be too short to accurately evaluate. Additionally, the value of the player's initial benefits increases by 10 every game from 70 to 170. This range could be expanded further. Finally, the functions to assign the value of cost, failure probability and player's benefit can be modified so that different results could be obtained.

Chapter 7

Conclusion and Future Work

This project has provided the empirical evaluation of the results of the ratios between the social utility of a pure strategy Nash equilibrium profile found by the algorithm suggested by [8] and the social utility of the optimal solution, in addition to the software implemented from the algorithm and the optimal solution finder. The results show that the upper bound of the ratio under the parameter settings during this experiment is approximately 2.41, implying that the quality of the obtained equilibrium computed by the algorithm can be 2.41 times as inefficient as the optimal solution. Furthermore, it suggests that the value 2.41 is the best possible value of the upper bound of the price of anarchy and the worst possible value of the upper bound of the price of stability in the CGLF. By observing the results represented as a graph, it is found that the value of failure probability is most likely to affect the value of the ratio among the other parameters.

The future work would be to conduct theoretical research to find a function of calculating such a ratio based on parameters: the number of players, the number of resources, each player's benefit, cost function and failure probability function. With this function, the potential designer of a system incorporating the CGLF is able to adjust the value of the set of parameters that minimise the ratio so that the quality of the system is improved. Alternatively, it would be beneficial to design algorithms to produce the theoretical result on the price of anarchy and the price of stability of a given CGLF game. This result would contribute to a more robust evaluation of the quality of a pure strategy Nash equilibrium the algorithm produces.

References

[1] Robert Axelrod. Effective choice in the prisoner's dilemma. *Journal of conflict resolution*, 24(1):3–25, 1980.

[2] George Christodoulou and Elias Koutsoupias. The price of anarchy of finite congestion games. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 67–73, New York, NY, USA, 2005. Association for Computing Machinery.

[3] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *Annual symposium on theoretical aspects of computer science*, pages 404–413. Springer, 1999.

[4] Igal Milchtaich. Congestion games with player-specific payoff functions. *Games and economic behavior*, 13(1):111–124, 1996.

[5] Dov Monderer and Lloyd S Shapley. Potential games. *Games and economic behavior*, 14(1):124–143, 1996.

[6] World Health Organization et al. Who director-general's opening remarks at the media briefing on covid-19-11 march 2020, 2020.

[7] Martin J Osborne and Ariel Rubinstein. *A course in game theory.* MIT press, 1994.

[8] Michal Penn, Maria Polukarov, and Moshe Tennenholtz. Congestion games with load-dependent failures: Identical resources. *Games and Economic Behavior*, 67(1):156–173, 2009.

[9] Robert W Rosenthal. A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, 2(1):65–67, 1973.

[10] Tim Roughgarden. Algorithmic game theory. *Commun. ACM*, 53(7):78–86, jul 2010.

[11] Tim Roughgarden. Intrinsic robustness of the price of anarchy. *Journal of the ACM (JACM)*, 62(5):1–42, 2015.

[12] Tim Roughgarden and Eva Tardos. Introduction to the inefficiency of equilibria. *Algorithmic game theory*, 17:443–459, 2007.

[13] Yoav Shoham. Computer science and game theory. *Communications of the ACM*, 51(8):74–79, 2008.

Appendix A

Extra Information

A.1 Tables

The following tables are the results of the ratios between an obtained Nash equilibrium and the optimal solution of each game. These tables are then converted to three dimensional graphs in Chapter 6.

Player\Resource	2	3	4
2	1.13	1.24	1.00
3	1.44	1.14	1.21
4	1.38	1.36	1.15

Table A.1: Change of the ratio by the number of players and resources with the setting: player's benefit = 100, cost = 5, failure probability = 0.33

Benefit\Cost	0	1	2	3	4	5	6	7	8	9	10
70	1	1.25	1.29	1.11	1.27	1.36	1.55	1.00	1.00	1.00	1.00
80	1	1.19	1.22	1.51	1.15	1.32	1.39	1.56	1.80	1.00	1.00
90	1	1.15	1.15	1.38	1.11	1.25	1.29	1.41	1.57	1.77	1.00
100	1	1.12	1.45	1.29	1.58	1.15	1.29	1.32	1.43	1.57	1.75
110	1	1.10	1.37	1.25	1.45	1.11	1.18	1.25	1.34	1.45	1.58
120	1	1.08	1.31	1.18	1.36	1.09	1.14	1.27	1.28	1.36	1.46
130	1	1.07	1.26	1.16	1.29	1.50	1.11	1.17	1.30	1.30	1.38
140	1	1.06	1.23	1.50	1.27	1.41	1.09	1.14	1.19	1.25	1.32
150	1	1.05	1.20	1.43	1.20	1.35	1.54	1.12	1.16	1.28	1.27
160	1	1.04	1.18	1.38	1.18	1.30	1.46	1.10	1.14	1.18	1.30
170	1	1.03	1.16	1.33	1.16	1.25	1.39	1.57	1.12	1.16	1.20

Table A.2: Change of the ratio by players' benefit and resource cost with the setting: number of players = 4, number of resource = 4, failure probability = 0.33

Benefit\FP	0	0.09	0.17	0.23	0.29	0.33	0.38	0.41	0.44	0.47	0.50
70	1.00	1.00	1.26	1.28	1.31	1.36	1.42	1.49	1.63	1.72	1.82
80	1.00	1.00	1.19	1.20	1.22	1.32	1.44	1.49	1.55	1.61	1.68
90	1.00	1.16	1.14	1.15	1.17	1.25	1.28	1.31	1.43	1.47	1.52
100	1.00	1.13	1.11	1.11	1.13	1.15	1.17	1.20	1.24	1.32	1.35
110	1.00	1.11	1.08	1.08	1.10	1.11	1.13	1.16	1.19	1.22	1.25
120	1.00	1.09	1.06	1.06	1.07	1.09	1.73	1.88	2.05	1.18	1.20
130	1.00	1.07	1.05	1.05	1.43	1.50	1.59	1.70	1.83	1.98	2.17
140	1.00	1.06	1.04	1.03	1.36	1.41	1.49	1.57	1.67	1.79	1.93
150	1.00	1.05	1.03	1.27	1.30	1.35	1.41	1.51	1.65	1.75	1.86
160	1.00	1.04	1.02	1.23	1.25	1.30	1.38	1.48	1.55	1.64	1.73
170	1.00	1.03	1.01	1.20	1.22	1.25	1.36	1.42	1.48	1.55	1.63

Table A.3: Change of the ratio by the number of players and resources with the setting: player's benefit = 100, cost = 5, failure probability = 0.33

Cost\FP	0	0.09	0.17	0.23	0.29	0.33	0.38	0.41	0.44	0.47	0.50
0	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1	1	1.04	1.11	1.10	1.11	1.12	1.14	1.16	1.18	1.20	1.23
2	1	1.00	1.10	1.11	1.11	1.45	1.52	1.59	1.69	1.80	1.95
3	1	1.04	1.02	1.22	1.25	1.29	1.40	1.46	1.53	1.60	1.69
4	1	1.08	1.06	1.06	1.07	1.58	1.68	1.81	1.97	2.16	1.19
5	1	1.13	1.11	1.11	1.13	1.15	1.17	1.20	1.24	1.32	1.35
6	1	1.00	1.17	1.17	1.19	1.29	1.41	1.45	1.50	1.55	1.61
7	1	1.00	1.23	1.24	1.28	1.32	1.37	1.50	1.57	1.77	2.06
8	1	1.00	1.30	1.33	1.37	1.43	1.51	1.60	1.70	1.82	1.97
9	1	1.00	1.00	1.43	1.49	1.57	1.68	1.81	1.97	2.16	2.41
10	1	1.00	1.00	1.00	1.63	1.75	1.91	2.10	1.00	1.00	1.00

Table A.4: Change of the ratio by resource cost and failure probability with the setting: number of players = 4, number of resource = 4, player's benefit = 100

Appendix B

User Guide

B.1 Instructions

In order to execute the system, users must have a Python3 environment in their computer and be in the software directory on command line. If users wish to obtain different data, they must be familiar with the basic operations of the Python language, especially the use of for loops.

B.1.1 Executing the Generation of Data

In order to obtain the data of the ratio between social utilities of a Nash equilibrium and the optimal solution, only the file ”main.py” must be executed. Users must follow the constraint mentioned in the main report. The following are the commands users must execute to obtain the data. If users' environment already contains ”openpyxl”, the first command does not have to be executed. It usually takes some time to generate data.

1. pip3 install openpyxl

2. python3 main.py

Testing

In order to conduct testing, users must execute the following command. With this command, all test files are executed.

1. python3 -m unittest discover tests

Change Parameters

If users wish to obtain data of the ratio with different parameter settings, it can be achieved with the following way. The figures B.1 and B.2 are a segment of code (from line 160 to 163 of main.py) that handle the parameter settings.

```
160  data_player_resource = [[calculate_ratio(player,resource,100,5,5) for resource in range(2,5)] for player in range(2,5)]
161  data_benefit_fp = [[calculate_ratio(4,4,benefit,5,fp) for fp in range(0,11)] for benefit in range(70,171,10)]
162  data_cost_fp = [[calculate_ratio(4,4,100,cost,fp) for fp in range(0,11)] for cost in range(0,11)]
163  data_benefit_cost = [[calculate_ratio(4,4,benefit,cost,5) for cost in range(0,11)] for benefit in range(70,171,10)]
```

Figure B.1: Parameter setting

```
170  data_player_resource = [[calculate_ratio(player,resource,100,5,5) for resource in range(2,5)] for player in range(2,5)]
171  data_benefit_fp = [[calculate_ratio(4,4,benefit,5,fp) for fp in range(11,21)] for benefit in range(0,10)]
172  data_cost_fp = [[calculate_ratio(4,4,100,cost,fp) for fp in range(0,11)] for cost in range(0,11)]
173  data_benefit_cost = [[calculate_ratio(4,4,benefit,cost,5) for cost in range(100,110)] for benefit in range(70,171,10)]
```

Figure B.2: Another parameter setting

The line 160 manipulates the parameters of player and resource, the line 161 manipulates the parameters of player's benefit and resource failure probability, the line 162 manipulates the parameters of resource cost and failure probability and the line 163 manipulates the parameters of player's benefit and resource cost.

Users can also change how player's benefit, resource cost and failure probability change by modifying the code from line 108 to 118 in main.py.

```
105      players = dict()
106      initial_benefit = benefit
107      for i in range(1, num_players + 1):
108          players[i] = Player(i, benefit)
109          benefit = initial_benefit - (i + 1) ** 2
110      failure_probability = dict()
111      for i in range(1, num_players + 1):
112          failure_probability[i] = 1 - 1 / (1 + i / 10 * start_probability)
113      cost = dict()
114      for i in range(1, num_players + 1):
115          cost[i] = i * (start_cost)
116      resources: Dict[int, Resource] = dict()
117      for i in range(1, num_resources + 1):
118          resources[i] = Resource(i, cost, failure_probability)
```

Figure B.3: Change the functions of parameters

Appendix C

Source Code

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary.

Hiro Funatsuka

April 8, 2022

The following are all the python files coded during this project. The test files belong to a folder ”tests”.

C.1 player.py

```python
class Player():

    def __init__(self, player_id: int, benefit: float):
        """
        Constructor that gets run when main.py is invoked

        Parameters
        ----------
        player_id : Dict[int, Player]
            player's id

        benefit : float
            player's benefit
        """

        self.__id = player_id
        self.__benefit = benefit

    def get_id(self) -> int:
        """
        Return player's id

        Returns
        -------
        id : int
            player's id
        """

        return self.__id

    def get_benefit(self) -> float:
        """
        Return player's benefit

        Returns
        -------
        float : float
            player's benefit
        """

        return self.__benefit

    def set_id(self, id: int):
        """
        Assign a new id to this player

        Parameters
        ----------
        id : int
            player's new id
        """

        self.__id = id

    def set_benefit(self, benefit: float):
        """
        Assign a new benefit to this player

        Parameters
```

```
        ----------
        benefit : float
            player's new benefit
        """

        self.__benefit = benefit
```

C.2 resource.py

```python
from typing import Dict

class Resource():

    def __init__(self, resource_id: int, cost: Dict[int, float], failure_probability: Dict[int, float]):
        """
        Constructor that gets run when main.py is invoked

        Parameters
        ----------
        resource_id : int
            resource's id

        cost : Dict[int, float]
            cost of this resource with congestion

        failure_probability : Dict[int, float]
            failure probability of this resource with congestion
        """

        self.__id = resource_id
        self.__costs: Dict[int, float] = cost
        self.__failure_probabilities = failure_probability

    def get_id(self) -> int:
        """
        Return resource's id

        Returns
        -------
        id : int
            resource's id
        """

        return self.__id

    def get_cost(self, congestion:int) -> float:
        """
        Return cost of resource given congestion

        Parameters
        ----------
        congestion : int
            congestion on this resource

        Returns
        -------
        costs : float
            cost
        """

        return self.__costs[congestion]

    def get_failure_probability(self, congestion:int) -> float:
        """
        Return failure probability given congestion

        Parameters
```

```
        ----------
        congestion : int
            congestion on this resource
        Returns
        -------
        failure_probabilities : float
            failure probability
        """

        return self.__failure_probabilities[congestion]

    def get_costs(self) -> Dict[int, float]:
        """
        Return costs of this resource

        Returns
        -------
        costs : Dict[int, float]
            costs
        """

        return self.__costs

    def get_failure_probabilities(self) -> Dict[int, float]:
        """
        Return failure probabilities of this resource

        Returns
        -------
        failure_probabilities : Dict[int, float]
            failure probabilities
        """

        return self.__failure_probabilities

    def set_id(self, resource_id: int):
        """
        Assign a new id to this resource

        Parameters
        ----------
        id : int
            resource's new id
        """

        self.__id = resource_id

    def set_cost(self, cost: float, congestion: int):
        """
        Assign a new cost to this resource at given congestion

        Parameters
        ----------
        cost : float
            resource's new cost

        congestion : int
            congestion
        """
```

```
        self.__costs[congestion] = cost

    def set_failure_probability(self, failure_probability: float, congestion:
int):
        """
        Assign a new failure probability to this resource at given congestion

        Parameters
        ----------
        failure_probability : float
            resource's new failure probability

        congestion : int
            congestion
        """

        self.__failure_probabilities[congestion] = failure_probability
```

C.3 strategy_ profile.py

```python
from copy import deepcopy

from player import Player
from resource import Resource

from typing import Dict, Set

class StrategyProfile():

    def __init__(self, strategies: Dict[int, Set[int]], players: Dict[int, Player], resources: Dict[int, Resource]):
        """
        Constructor that gets run when CGLF class construct strategy profiles or equilibrium profile is constructed

        Parameters
        ----------
        players : Dict[int, Player]
            a set of players in a game

        resources : Dict[int, Resource]
            a set of resources to be used by players in a game
        """

        self.__strategies: Dict[int, Set[int]] = strategies # key: player_id, value: a set of int(resource id)
        self.__players: Dict[int, Player] = players
        self.__resources: Dict[int, Resource] = resources
        self.__utilities: Dict[int, float] = dict() # key: player_id, value: float
        self.__congestion: Dict[int, int] = dict()
        self.__even: int = None
        self.__social_utility: float = None
        self.update_profile()

    def update_profile(self):
        """
        Update member variables
        """

        self.__congestion = self.calculate_congestion(self.__strategies)
        self.__even = self.check_even()
        self.calculate_utilities()
        self.__social_utility = sum(self.__utilities.values())

    def check_even(self) -> int:
        """
        Check whether this strategy profile's resouces are evenly assigned to players

        Returns
        -------
        congestion : int
            common congestion number
        """

        congestions = set()
        for congestion in self.__congestion.values():
            congestions.add(congestion)
```

```
        if len(congestions) == 1:
            return congestions.pop()
        else:
            return None

    def get_utilities(self) -> Dict[int, float]:
        """
        Return a member variable of utilities of all players

        Returns
        -------
        utilities : Dict[int, float]
            utilities of all players, key: player_id, value: utility value
        """

        return self.__utilities

    def get_strategies(self) -> Dict[int, Set[int]]:
        """
        Return a member variable of strategies of all players

        Returns
        -------
        strategies : Dict[int, Set[int]]
            strategies of all players, key: player_id, value: set of resource_id
        """

        return self.__strategies

    def get_utility(self, player_id) -> float:
        """
        Return a member variable of utility of player

        Parameters
        ----------
        player_id : int
            player's id

        Returns
        -------
        utility : float
            player's utility
        """

        return self.__utilities[player_id]

    def get_social_utility(self) -> float:
        """
        Return a member variable of social_utility

        Returns
        -------
        social_utility : float
            social utility of this strategy profile
        """

        return self.__social_utility
```

```python
        """
        Return a member variable of congestion

        Returns
        -------
        congestion : Dict[int, int]
            a dict of congestion (key: resource_id, value: amount of congestion)
        """

        return self.__congestion

    def calculate_congestion(self, strategies: Dict[int, Set[int]]) -> Dict[int, int]:
        """
        Calculate congestion given strategies of all players

        Parameters
        ----------
        strategies : Dict[int, Set[int]]
            strategies of all players, key: player_id, value: set of resource_id

        Returns
        -------
        congestion : Dict[int, int]
            congestion, key: resource_id, congestion
        """

        congestion = {key: 0 for key in self.__resources.keys()} # key: resource_id, value: int
        for strategy in strategies.values():
            for resource in strategy:
                congestion[resource] += 1
        return congestion

    def simulate_change(self, strategy: Set[int], player_id: int) -> bool:
        """
        Simulate the if a given player change to another strategy in this profile

        Parameters
        ----------
        strategy : Set[int]
            a set of resources to be used by players in a game

        player_id : int
            player's id

        Returns
        -------
        strategy_set : bool
            Return true if the change is beneficial to the player; otherwise, false
        """

        new_strategies = deepcopy(self.__strategies)
        new_strategies[player_id] = strategy
        new_congestion = self.calculate_congestion(new_strategies)
```

```
        return self.calculate_utility(player_id, strategy, new_congestion) > self.__utilities[player_id]

    def calculate_utility(self, player_id: int, strategy: Set[int], congestion: Dict[int, int]) -> float:
        """
        Calculate a utility of a given player

        Parameters
        ----------
        player_id : int
            player's id

        strategy : Set[int]
            player's strategy

        congestion : Dict[int, int]
            congestion of a strategy profile

        Returns
        -------
        strategy_set : float
            player's utility
        """

        probability_product = 1 # if player didn't choose any resource, failure probability = 1
        total_cost = 0
        for resource in strategy:
            failure_probability = self.__resources[resource].get_failure_probability(congestion[resource])
            cost = self.__resources[resource].get_cost(congestion[resource])
            probability_product *= failure_probability
            total_cost += cost
        return self.__players[player_id].get_benefit()*(1-probability_product) - total_cost

    def calculate_utilities(self):
        """
        Calculate utilities of all players
        """

        for player_id, strategy in self.__strategies.items():
            self.__utilities[player_id] = self.calculate_utility(player_id, strategy, self.__congestion)

    def display_result(self):
        """
        Print the information of this strategy profile
        """

        print(f'Number of players: {len(self.__players.keys())}')
        print(f'Social Utility: {self.__social_utility}')
        print(f'Resource Cost: {self.__resources[1].get_costs()}')
        print(f'Resource Failure Probability: {self.__resources[1].get_failure_probabilities()}')
        print()
        for player_id in self.__players.keys():
            print(f'Player {player_id}')
```

```
            print(f'Strategy: {self.__strategies[player_id]}')
            print(f'Utility: {self.__utilities[player_id]}')
            print()
            print()
```

C.4 equilibrium.py

```python
from strategy_profile import StrategyProfile
from copy import deepcopy
from typing import Dict, List, Set
from player import Player
from resource import Resource

# Code modified from the algorithm obtained from https://doi.org/10.1016/j.geb.2009.03.004
class Equilibrium():

    def __init__(self, players: Dict[int, Player], resources: Dict[int, Resource]):
        """
        Constructor that gets run when main.py is invoked

        Parameters
        ----------
        players : Dict[int, Player]
            a set of players in a game

        resources : Dict[int, Resource]
            a set of resources to be used by players in a game
        """

        self.__k: int = len(players)
        self.__xD: Dict[int,int] = dict()
        self.__xA: Dict[int,int] = dict()
        self.__x: Dict[int,int] = dict()

        self.__players: Dict[int, Player] = players
        self.__resources: Dict[int, Resource] = resources

        self.__profile: StrategyProfile = self.step0()

    def calculate_marginal_benefit(self, player: Player, congestion: int, number_of_resources: int) -> float:
        """
        Calculate a marginal benefit for the player given the congestion and the number of resources under k-even strategy profile

        Parameters
        ----------
        resources : Dict[int, Resource]
            a set of resources to be used by players in a game

        player : Player
            a player of whom marginal benefit is calculated

        congestion : int
            integer value

        number_of_resources : int
            the number of resources

        Returns
        -------
        marginal benefit : float
            marginal benefit
        """
```

```python

        return player.get_benefit() * (self.__resources[1].get_failure_probability(congestion)**(number_of_resources))

    def calculate_marginal_cost(self, congestion: int) -> float:
        """
        Calculate marginal cost given the congestion under k-even strategy profile

        Parameters
        ----------
        congestion : int
            congestion for the resource

        Returns
        -------
        marginal cost : float
            marginal cost
        """

        return self.__resources[1].get_cost(congestion)/(1-self.__resources[1].get_failure_probability(congestion))

    def sigma(self, start: int, end: int, collection: dict) -> int:
        """
        Calculate the sum of collection. If the value of collection is set, the sum of the lengths of set is calculated;
        otherwise, the value is an integer and the sum of intergers is calculated

        Parameters
        ----------
        start : int
            start index of collection to calculate

        end : int
            end index of collection to calculate

        collection : dict
            collection of which sum is calculated

        Returns
        -------
        sum : int
            sum
        """

        sum = 0
        for i in range(start, end + 1):
            if type(collection[i]) == set:
                sum += len(collection[i])
            elif type(collection[i]) == int:
                sum += collection[i]
        return sum

    def get_equilibrium_profile(self) -> StrategyProfile:
        """
        Return a member variable of a equilibrium strategy profile
```

```python
        Returns
        -------
        profile : StrategyProfile
            equilibrium strategy profile
        """

        return self.__profile

    def step0(self) -> StrategyProfile:
        """
        Be called inside constructor, calling this function explores all the other steps to find an equilibrium profile if needed.
        Check D-stability at a n-even profile

        Returns
        -------
        strategy profile : StrategyProfile
            equilibrium profile
        """

        player: Player = min(self.__players.values(), key=lambda x:x.get_benefit())
        if self.calculate_marginal_benefit(player, len(self.__players), len(self.__resources)-1) >= self.calculate_marginal_cost(len(self.__players)): # player with the lowest benefit
            return StrategyProfile({key:set(self.__resources.keys()) for key in self.__players.keys()}, self.__players, self.__resources)
        else:
            self.__k -= 1
            return self.step1()

    def step1(self):
        """
        Determine the value of k
        """

        for player in self.__players.values():
            XD = []
            for x in range(1, len(self.__resources)+1):
                if self.calculate_marginal_benefit(player, self.__k, x-1) >= self.calculate_marginal_cost(self.__k):
                    XD.append(x)
            if XD == []:
                self.__xD[player.get_id()] = 0
            else:
                self.__xD[player.get_id()] = max(XD)

        if sum(self.__xD.values()) < self.__k * len(self.__resources):
            self.__k -= 1
            return self.step2()
        else:
            return self.step3()

    def step2(self):
        """
        Construct an equilibrium profile if k is confirmed to be 0
        """
```

```
            strategies = dict()
            resource_index = 1
            for player in self.__players.values():
                if self.__xD[player.get_id()] > 0:
                    strategy = set()
                    for i in range(resource_index, resource_index + self.__xD[player.get_id()]):
                        strategy.add(self.__resources[i].get_id())
                    resource_index += self.__xD[player.get_id()]
                    strategies[player.get_id()] = strategy
                else:
                    strategies[player.get_id()] = set()
            return StrategyProfile(strategies, self.__players, self.__resources)
        else:
            return self.step1()

    def step3(self):
        """
        Check the existence of a k-even equilibrium profile
        """

        for player in self.__players.values():
            XA = []
            for x in range(len(self.__resources)):
                if self.calculate_marginal_benefit(player, self.__k, x) <= self.calculate_marginal_cost(self.__k+1):
                    XA.append(x)

            if XA == []:
                self.__xA[player.get_id()] = len(self.__resources)
            else:
                self.__xA[player.get_id()] = min(XA)

        if sum(self.__xA.values()) > self.__k * len(self.__resources) or any(self.__xA[player_id] > self.__xD[player_id] for player_id in self.__players.keys()):
            return self.step5()
        else:
            return self.step4()

    def step4(self):
        """
        Construct a k-even equilibrium profile
        """

        d: Dict[int, int] = dict()
        strategies = {key:set() for key in self.__players.keys()}
        resource_index = 1
        for i in range(1, len(self.__players) + 1):
            d[i] = self.__k * len(self.__resources) - self.sigma(1, i - 1, strategies) - self.sigma(i, len(self.__players), self.__xA)
            r = min([self.__xD[i], self.__xA[i] + d[i]])
            if resource_index == len(self.__resources):
                resource_index = 1
            strategy = set()

            while r > 0:
                if resource_index == 0:
```

```
                strategy.add(resource_index)
                resource_index = (resource_index + 1) % len(self.__resources)
                r -= 1
            strategies[self.__players[i].get_id()] = strategy
        return StrategyProfile(strategies, self.__players, self.__resources)

    def step5(self):
        """
        Construct a post-addition D-stable profile
        """

        for player in self.__players.values():
            X = []
            for x in range(1, len(self.__resources)):
                if self.calculate_marginal_benefit(player, self.__k, x - 1) >= self.calculate_marginal_cost(self.__k + 1):
                    X.append(x)
            if X == []:
                self.__x[player.get_id()] = 0
            else:
                self.__x[player.get_id()] = max(X)

        resource_index = 1
        strategies = {key:set() for key in self.__players.keys()}
        for i in range(1, len(self.__players) + 1):
            delta = self.__k * len(self.__resources) - self.sigma(1, i - 1, strategies)
            if delta > 0:
                r = min([self.__x[i], delta])
                strategy = set()
                if resource_index == 0:
                    resource_index = len(self.__resources)
                while r > 0:
                    if resource_index == 0:
                        resource_index = len(self.__resources)
                    strategy.add(resource_index)
                    resource_index = (resource_index + 1) % len(self.__resources)
                    r -= 1
                strategies[i] = strategy
            else:
                strategies[i] = set()

        return self.step6(StrategyProfile(strategies, self.__players, self.__resources))

    def step6(self, strategy_profile):
        """
        Check A-stability

        Parameters
        ----------
        strategy_profile : StrategyProfile
            k*-even D-stable strategy profile
        """

        a_move_resources: Dict[int,int] = dict()
        for player in self.__players.values():
            strategies = strategy_profile.get_strategies()
```

```python
            option = {key:value for key, value in strategy_profile.get_congestion().items() if not key in strategies[player.get_id()]}
            light_resource = None
            if len(option) > 0:
                light_resource = min(option, key=option.get)
            new_strategy = deepcopy(strategies[player.get_id()])
            new_strategy.add(light_resource)
            if light_resource != None and strategy_profile.simulate_change(new_strategy, player.get_id()):
                a_move_resources[player.get_id()] = light_resource

        if len(a_move_resources) == 0:
            return strategy_profile
        else:
            return self.step7(strategy_profile, a_move_resources)

    def step7(self, strategy_profile, a_move_resources):
        """
        Conduct a one- or two-step addition

        Parameters
        ----------
        strategy_profile : StrategyProfile
            k*-even D-stable strategy profile

        a_move_resources : Dict[int, int]
            a collection of resources which players get benefits to add
        """

        a_move_player, light_resource_a = min(a_move_resources.items(), key=lambda x: x[1])
        strategies = strategy_profile.get_strategies()

        if light_resource_a == min(strategy_profile.get_congestion().items(), key=lambda x: x[1])[0]:
            strategies[a_move_player].update({light_resource_a})
        else:
            light_resource_b = min(strategy_profile.get_congestion(), key=strategy_profile.get_congestion().get)
            player_j = [key for key, value in strategies.items() if light_resource_a in value and not light_resource_b in value][0]
            strategies[a_move_player].update({light_resource_a})
            strategies[player_j].update({light_resource_b})
        strategy_profile.update_profile()

        return self.step6(strategy_profile)
```

C.5 cglf.py

```python
import itertools
from strategy_profile import StrategyProfile
from typing import Dict, List, Set
from player import Player
from resource import Resource


class CGLF():

    def __init__(self, players: Dict[int, Player], resources: Dict[int, Resource]):
        """
        Constructor that gets run when main.py is invoked

        Parameters
        ----------
        players : Dict[int, Player]
            a set of players in a game

        resources : Dict[int, Resource]
            a set of resources to be used by players in a game
        """

        self.players: Dict[int, Player] = players
        self.resources: Dict[int, Resource] = resources

        self.__strategy_set: List[Set[Resource]] = self.set_strategy_set(resources)
        self.__strategy_profiles: List[StrategyProfile] = self.build_strategy_profiles()
        self.__optimal_profile: StrategyProfile = max(self.__strategy_profiles, key=lambda x:x.get_social_utility())


    def set_strategy_set(self, resources: Dict[int, Resource]) -> List[Set[Resource]]:
        """
        Create a strategy set given resources

        Parameters
        ----------
        resources : Dict[int, Resource]
            a set of resources to be used by players in a game

        Returns
        -------
        strategy_set : List[Set[Resource]]
            strategy set
        """

        strategy_set = []
        for i in range(len(resources)+1):
            for strategy in itertools.combinations(resources, i):
                strategy_set.append(set(list(strategy)))
        return strategy_set

    # Code modified from https://qiita.com/mSpring/items/c4973ea214a36c4a699c
    def build_strategy_profiles(self) -> List[StrategyProfile]:
        """
```

```
        Create strategy profiles given players and resources

        Returns
        -------
        strategy_profiles : List[StrategyProfile]
            strategy profiles
        """

        strategy_sets = dict()
        for player in self.players.values():
            strategy_sets[player.get_id()] = self.__strategy_set
        product = [x for x in itertools.product(*strategy_sets.values())]
        strategies_set = [dict(zip(strategy_sets.keys(), r)) for r in product]
        strategy_profiles = []
        for strategies in strategies_set:
            strategy_profiles.append(StrategyProfile(strategies, self.players, self.resources))
        return strategy_profiles

    def get_strategy_profiles(self) -> List[StrategyProfile]:
        """
        Return a member variable of strategy profiles

        Returns
        -------
        strategy_profiles : List[StrategyProfile]
            strategy profiles
        """

        return self.__strategy_profiles

    def get_optimal_profile(self) -> StrategyProfile:
        """
        Return a member variable of strategy profile

        Returns
        -------
        optimal_profile : StrategyProfile
            optimal strategy profile
        """

        return self.__optimal_profile

    def display_all(self):
        """
        Print data of all strategy profiles
        """

        for sp in self.__strategy_profiles:
            for player_id in sp.players.keys():
                print(f'Player {player_id}')
                print(f'Resource: {sp.strategies[player_id]}')
                print(f'Utility: {sp.utilities[player_id]}')
                print()
            print()
            print()
```

C.6 main.py

```python
from resource import Resource
from player import Player
from cglf import CGLF
from equilibrium import Equilibrium
from strategy_profile import StrategyProfile
from typing import Dict

import openpyxl


def validation(num_players, num_resources, benefit, start_cost, start_probability):
    if not type(num_players) == int or (num_players < 2 or num_players > 10):
        print(f'The number of players must be more than 1 and less than 11.')
        return False
    if not type(num_resources) == int or (num_resources < 2 or num_resources > 10):
        print(f'The number of players must be more than 1 and less than 11.')
        return False
    if not type(benefit) == int or benefit < 0:
        print(f'Benefit must be non-negative number.')
        return False
    if not type(start_cost) == int or start_cost < 0:
        print(f'Cost must be non-negative number.')
        return False
    if not type(start_probability) == int:
        return False
    return True

def check_algorithm(num_players, num_resources, benefit, start_cost, start_probability):
    """
    Check whether the software compute a correct Nash equilibrium

    Parameters
    ----------
    num_players : int
        the number of players for this game

    num_resources : int
        the number of resources for this game

    benefit : int
        initial player's benefit for this game

    start_cost : int
        initial cost of resource for this game

    start_probability : int
        initial failure probability of resource for this game
    """

    if not validation(num_players, num_resources, benefit, start_cost, start_probability):
        return False
    players = dict()
    initial_benefit = benefit
    for i in range(1, num_players + 1):
        players[i] = Player(i, benefit)
```

```
        benefit = initial_benefit - (i + 1) ** 2
    failure_probability = dict()
    for i in range(1, num_players + 1):
        failure_probability[i] = 1 - 1 / (1 + i / 10 * start_probability)
    cost = dict()
    for i in range(1, num_players + 1):
        cost[i] = i * (start_cost)
    resources: Dict[int, Resource] = dict()
    for i in range(1, num_resources + 1):
        resources[i] = Resource(i, cost, failure_probability)
    cglf = CGLF(players, resources)
    optimal_profile: StrategyProfile = cglf.get_optimal_profile()
    optimal_profile.display_result()
    #cglf.display_all()
    equilibrium_profile: StrategyProfile = Equilibrium(players, resources).get_equilibrium_profile()
    equilibrium_profile.display_result()
    print(optimal_profile.get_social_utility()/equilibrium_profile.get_social_utility())

def calculate_ratio(num_players, num_resources, benefit, start_cost, start_probability):
    """
    Calculate the ratio between social utilities of an obtained Nash equilibrium and
    the optimal solution of this game

    Parameters
    ----------
    num_players : int
        the number of players for this game

    num_resources : int
        the number of resources for this game

    benefit : int
        initial player's benefit for this game

    start_cost : int
        initial cost of resource for this game

    start_probability : int
        initial failure probability of resource for this game

    Returns
    -------
    ratio : float
        inefficiency of an obtained Nash equilibrium of this game
    """

    if not validation(num_players, num_resources, benefit, start_cost, start_probability):
        return False

    players = dict()
    initial_benefit = benefit
    for i in range(1, num_players + 1):
        players[i] = Player(i, benefit)
```

```python
    failure_probability = dict()
    for i in range(1, num_players + 1):
        failure_probability[i] = 1 - 1 / (1 + i / 10 * start_probability)
    cost = dict()
    for i in range(1, num_players + 1):
        cost[i] = i * (start_cost)
    resources: Dict[int, Resource] = dict()
    for i in range(1, num_resources + 1):
        resources[i] = Resource(i, cost, failure_probability)
    cglf = CGLF(players, resources)
    optimal_profile: StrategyProfile = cglf.get_optimal_profile()
    equilibrium_profile: StrategyProfile = Equilibrium(players, resources).get_equilibrium_profile()
    social_optima: float = optimal_profile.get_social_utility()
    if int(equilibrium_profile.get_social_utility()) != 0:
        ratio = social_optima / equilibrium_profile.get_social_utility()
        return ratio
    else:
        print("The social utility of the optimal solution was 0. The ratio was not calculated.")
        return None

# Code modified from https://himibrog.com/python-output-excel/
def export_excel_2d(data, index):
    """
    Export an Excel file given an input

    Parameters
    ----------
    data : List[List[float]]
        a set of data

    index : int
        used for naming a file
    """

    wb = openpyxl.Workbook()

    # Check Sheet
    print(f'Sheet name: {wb.get_sheet_names()}')

    # Retrieve sheet object
    s1 = wb.get_sheet_by_name(wb.get_sheet_names()[0])

    for i in range(len(data)):
        for j in range(len(data[i])):
            s1.cell(row=i+1,column=j+1,value=data[i][j])

    wb.save(f'data_{index}_.xlsx')

#check_algorithm(4, 4, 1000,10,5)

data_player_resource = [[calculate_ratio(player,resource,100,5,5) for resource in range(2,5)] for player in range(2,5)]
data_benefit_fp = [[calculate_ratio(4,4,benefit,5,fp) for fp in range(0,11)] for benefit in range(70,171,10)]
data_cost_fp = [[calculate_ratio(4,4,100,cost,fp) for fp in range(0,11)] for cost in range(0,11)]
data_benefit_cost = [[calculate_ratio(4,4,benefit,cost,5) for cost in
```

```
dataset = [data_player_resource, data_benefit_fp, data_cost_fp,
data_benefit_cost]

for i in range(len(dataset)):
    export_excel_2d(dataset[i], i)
```

C.7 test_ strategy_ profile.py

```python
from unittest import TestCase
from player import Player
from resource import Resource
from strategy_profile import StrategyProfile

class StrategyProfileTest(TestCase):
    def setUp(self):
        player1 = Player(1,1.1)
        player2 = Player(2,4)
        self.__players = {player1.get_id():player1,player2.get_id():player2}
        cost = {1:1,2:2}
        failure_probability = {1:0.01,2:0.26}
        resource1 = Resource(1, cost, failure_probability)
        resource2 = Resource(2, cost, failure_probability)
        self.__resources = {resource1.get_id():resource1, resource2.get_id():resource2}
        self.__strategies = {1:{1},2:{1,2}}
        self.__profile = StrategyProfile(self.__strategies, self.__players, self.__resources)

    def test_check_even(self):
        result = self.__profile.check_even()
        expected = None
        self.assertEqual(expected, result)

    def test_get_congestion(self):
        result = self.__profile.get_congestion()
        expected = {1:2,2:1}
        self.assertEqual(expected, result)

    def test_simulate_change(self):
        result = self.__profile.simulate_change({1,2}, 1)
        expected = False
        self.assertEqual(expected, result)

    def test_calculate_utility(self):
        result = self.__profile.calculate_utility(2, self.__profile.get_strategies()[2], self.__profile.get_congestion())
        expected = 4 * (1 - 0.01 * 0.26) - (1 + 2)
        self.assertEqual(result, expected)

    def tearDown(self):
        del self.__profile
```

C.8 test_ cglf.py

```python
from unittest import TestCase
from cglf import CGLF
from player import Player
from resource import Resource
from strategy_profile import StrategyProfile

class CGLFTest(TestCase):
    def setUp(self):
        player1 = Player(1,1.1)
        player2 = Player(2,4)
        self.players = {player1.get_id():player1,player2.get_id():player2}
        cost = {1:1,2:2}
        failure_probability = {1:0.01,2:0.26}
        resource1 = Resource(1, cost, failure_probability)
        resource2 = Resource(2, cost, failure_probability)
        self.resources = {resource1.get_id():resource1, resource2.get_id():resource2}
        self.cglf = CGLF(self.players, self.resources)

    def test_set_strategy_set(self):
        result = self.cglf.set_strategy_set(self.resources)
        expected = [set(), {1}, {2}, {1,2}]
        self.assertEqual(result, expected)

    def test_get_optimal_profile(self):
        result = result = self.cglf.get_optimal_profile().get_social_utility()
        expected = StrategyProfile({1:{1}, 2:{2}}, self.players, self.resources).get_social_utility()

    def test_build_strategy_profiles(self):
        profiles = self.cglf.build_strategy_profiles()
        result = {profile.get_social_utility() for profile in profiles}
        expected = {StrategyProfile({1:{}, 2:{}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{1}, 2:{}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{2}, 2:{}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{1,2}, 2:{}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{}, 2:{1}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{1}, 2:{1}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{2}, 2:{1}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{1,2}, 2:{1}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{}, 2:{2}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{1}, 2:{2}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{2}, 2:{2}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{1,2}, 2:{2}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{}, 2:{1,2}}, self.players, self.resources).get_social_utility(),
```

```
                    StrategyProfile({1:{1}, 2:{1,2}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{2}, 2:{1,2}}, self.players, self.resources).get_social_utility(),
                    StrategyProfile({1:{1,2}, 2:{1,2}}, self.players, self.resources).get_social_utility()}
        self.assertEqual(result, expected)


    def tearDown(self):
        del self.cglf
```

C.9 test_ equilibrium.py

```python
from unittest import TestCase
from player import Player
from resource import Resource
from equilibrium import Equilibrium

class EquilibriumTest(TestCase):
    def setUp(self):
        player1 = Player(1,4)
        player2 = Player(2,1.1)
        self.players = {player1.get_id():player1,player2.get_id():player2}
        cost = {1:1,2:2}
        failure_probability = {1:0.01,2:0.26}
        resource1 = Resource(1, cost, failure_probability)
        resource2 = Resource(2, cost, failure_probability)
        self.resources = {resource1.get_id():resource1, resource2.get_id():resource2}
        self.equilibrium = Equilibrium(self.players, self.resources)

    def test_step0(self):
        result = self.equilibrium.get_equilibrium_profile()
        expected = 2.96 + 0.089
        self.assertEqual(result.get_social_utility(), expected)

    def test_sigma_1(self):
        start = 1
        end = 0
        d = {1:{1,2,3},2:{4,5},3:{1}}
        result = self.equilibrium.sigma(start, end, d)
        self.assertEqual(result, 0)

    def tearDown(self):
        del self.equilibrium
```