Algorithms for computing clearing payments in financial networks

Kexin Liu

4th Year Project Report
Artificial Intelligence
School of Informatics
University of Edinburgh

2023

Abstract

The clearing problem, highlighted by Eisenberg and Noe in [9], aims to calculate the clearing payment for all financial institutions in the event of cascading default. Solving the clearing problem is complex due to the intricate nature of real-world financial systems. However, Eisenberg and Noe proposed a general financial model with only debt contracts, and provided the Fictitious Default Algorithm to solve the clearing problem in this financial system, assuming that cascading default occurs round by round.

In this project, three different algorithms inspired by [9] were implemented in Python to compute the optimum clearing payment. These algorithms included one based on linear programming, one based on value iteration, and one utilizing the Fictitious Default Algorithm. Due to the scarcity of existing data sets, reasonable random data was generated based on the features of the financial network to conduct experiments and compare the algorithms' performance.

The results showed that all three algorithms had their strengths and weaknesses, and each was more suitable under certain conditions. Linear programming gave the optimum solution for most of the time, value iteration provided the fastest speed, and the Fictitious Default Algorithm allowed us to trace the path of contagion while maintaining similar speed with value iteration.

In addition, this project introduced an extended model that incorporated equity contracts and bankruptcy costs, based on [14]. We also implemented the algorithm provided by the authors to determine the equilibrium banks' value, which can be used to easily find the clearing payment vector after establishing this equilibrium value. We then analyzed the extended model by conducting experiments using randomly generated data and assessed its computing time as well. The results show that the computational efficiency becomes significantly slower when the number of banks increases, particularly when reaching about 300.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Kexin Liu*)

Acknowledgements

Thanks to my supervisor Professor Kousha Etessami for his advice and guidance throughout the year.

Thanks to my parents, my grandparents, my little sister and all my loving relatives, giving me both financial support and lots of love from the other side of the world.

Thanks to those friends who actively reach out to me, who listen to me, who accompany me and work with me... Thanks for all those moments filled with laughter and healing.

Thanks to dancing and the friends who dance with me, I couldn't imagine my life without all those moments.

Thanks to all the sunshine in Edinburgh, thanks to the Appleton Tower for the wonderful view, thanks to all the beautiful things I met.

At last, thanks to myself for all the hard work I've put in over these four years, persevering through all the tough times.

Contents

Chapter 1

Introduction

1.1 Initiative

The financial market has always been a hotspot in the world economy. Its vigorous development allows financial institutions, such as firms and banks, to engage in various financial contracts with each other, leading to complex financial networks with extensive interconnections. Although these contracts may help firms mitigate idiosyncratic liquidity variations or achieve superior investments, it is a fact that "financial interdependencies generate systemic risks" [16]. Indeed, Jackson *et al.* discuss two types of systemic risks in their survey: contagion through network interdependencies, and a canonical form characterized by a cascade of insolvencies (defaults).

Firms are interconnected by obligations, and if one such firm defaults unexpectedly, it can reduce the inflows of the firms that are its creditors, triggering them into default as well, consequently affecting even more firms. This kind of cascading failure could result in an economic downturn, as seen in the worldwide financial crisis of 2008, which was triggered by the collapse of the investment bank Lehman Brothers and affected the global financial system, leading to the worst recession since the Great Depression. Recently, the insolvency of Silicon Valley Bank and the subsequent failure of another US lender, Signature Bank, has sparked global fears of a systemic crisis and domino effects like those seen in the 2008 financial crisis.

Therefore, it is necessary to predict and measure this type of systemic risk in financial networks, as well as to compute the clearing payment (i.e., the payments needed to settle all debts between financial institutions in a network) in the event of cascading defaults. The latter is called the clearing problem, which was first highlighted in 2001 by Eisenberg and Noe [9]. The clearing problem is a computational challenge due to the complexity of financial systems. However, Eisenberg and Noe provided a general financial model of a clearing system that satisfies the standard conditions of bankruptcy law and presented an algorithm to compute the clearing payment vector (i.e., the solution of the clearing problem, which is a vector of clearing payments from firms to other firms in a financial system) for a financial system with only debt contracts in polynomial time. Building upon the model and algorithm given by Eisenberg and Noe, many other researchers have introduced additional financial contracts for clearing problems, such

as credit default swaps, which have been derived and proven to be a PPAD-complete problem [18].

These algorithms are crucial for actors in global finance to facilitate the analysis of systemic risk, rescue a financial system in crisis and prevent cascading defaults from happening again by scrutinizing financial institutions and analysing the stability of the financial system. For instance, in the case of bailing out banks to prevent contagion defaults, around three-quarters of the 104 bank failures studied by [11] involved a bailout in some form or another. However, real-life financial systems are too intricate, making the problem of systemic risk challenging. Therefore, the initiative of this project is to start with the financial model established by Eisenberg and Noe, investigate the algorithms for finding the clearing payment, and consider more complicated financial models thereafter.

1.2 Objectives

Although the theoretical frameworks for computing clearing vectors using the algorithms proposed by Eisenberg and Noe exist, to our knowledge, no one has yet implemented them. In this project, we will begin by implementing the three algorithms proposed and inspired by Eisenberg and Noe in [9], including one explicitly outlined in the paper and two inspired by the definitions provided. We will use a financial model that assumes only debt contracts exist in the financial system and no bankruptcy costs. As more recent papers have pointed out, bankruptcy costs are non-negligible and occur when firms default due to activities like a fire sale. Therefore, we will use another model provided by Jackson *et al.*, which adds equity contracts and bankruptcy costs to the model, and implement their algorithm as well [14]. Finally, using Python, we will experiment with all four algorithms, generating data according to the restrictions of the model and empirical knowledge of real financial networks, and analyse and compare the results of these algorithms.

1.3 Report Outline

In the upcoming chapter, we will introduce the mathematical definitions that serve as the basis of the further work. To improve the flow, we will present two financial systems and their corresponding algorithms in separate chapters. Chapter 3 will start by formalizing the financial system mentioned by Eisenberg and Noe and provide a detailed explanation of the implemented algorithms, including a thorough analysis of their complexity and accompanying pseudocode [9]. We will then delve into the specific implementation details of these algorithms, including the technologies employed. The last section of the chapter will cover the experiments carried out, including methods for data initialization and result analysis. In Chapter 4, we will apply the same approach to the financial system and algorithm mentioned by Jackson *et al.* [14]. Finally, the project will conclude with a summary of key findings and proposed open problems for future research and development.

Chapter 2

Mathematical Definitions

The algorithms utilized in this project rely on certain mathematical definitions and theorems. To better understand these algorithms and their analysis, we introduce some definitions related to lattices.

2.1 Complete Lattice Theory

A complete lattice (L,$\leq$) is a partially ordered set (or poset in short) in which every subset of L has both a least upper bound (or supremum) and a greatest lower bound (or infrimum) [4].

2.2 Lattice Operations

Three lattice operations need to clarified at the beginning to avoid ambiguities:

$$x \wedge y := (min[x_1, y_1], \quad min[x_2, y_2] ... min[x_n, y_n]),$$

$$x \vee y := (max[x_1, y_1], \quad max[x_2, y_2] ... max[x_n, y_n]),$$

$$x^+ := (max[x_1, 0], \quad max[x_2, 0] ... max[x_n, 0]).$$

2.3 Tarski's Fixed Point Theorem

Consider a complete lattice (L, $\leq$). Suppose there is a monotonic (increasing) function $f : L \to L$ (i.e., for all $x, y \in L$, $x \leq y$ implies $f(x) \leq f(y)$). Then, the set P, which is the set of all fixed points of f, is non-empty and is a complete lattice with respect to $\leq$ (i.e. (P, $\leq$)).
Consequently, f has a greatest fixed point and a least fixed point [19].

Chapter 3

Financial Model with Debt Contracts Only

In this chapter, we will focus on the financial model and the algorithms introduced in [9].

3.1 The Financial Model of Eisenberg and Noe

Cyclical interdependence is an important feature in financial system architectures, which was termed by Eisenberg and Noe in their paper 'Systemic Risk in Financial Systems' in 2001 [9]. This interdependence arises from the fact that the values of many firms are contingent on payments received from other firms. If one firm, say firm A, defaults, it may lead to firm B defaulting, as firm B will not receive the expected payoffs from firm A. This, in turn, could lead to firm C defaulting for the same reason. Consequently, firm C may be unable to complete its obligations to firm A. This paper pays attention to this linkage and develops a financial system model with this type of linkage that satisfies the standard conditions of bankruptcy law, including three important criteria. The first criterion is limited liability, which means that a firm should never pay more than it has in cash flow. The second criterion is that a firm should not pay any of its stockholders until it has paid off all its outstanding liabilities to the other firms, that is, the debt has absolute priority over equity. The third criterion is proportionality. When a firm defaults, it may not be able to fulfil all of its obligations to other firms, so it should pay the claimant firms according to the proportion of the firms' nominal claim to the sum of the claim, so that no firm feels unfairly treated during the clearing. Following these conditions, this project provides a simple and tractable model for computing clearing vectors, which is the output vector that the clearing mechanism tries to compute - payments from firms to other firms in the financial system.

To formalize the model, better illustrate the algorithm, and avoid any ambiguities, it is necessary to introduce some key definitions and notations used in [9] to construct the financial system model.

Consider a financial network with n nodes, each of which represents a distinct economic entity or financial node, and has nominal liabilities(i.e. promised payments) to other nodes in the system, then:

- Each node (a bank, a firm etc.) is represented by a node n.
- L_{ij} is the promised payment value from node i to node j, referred to as nominal liability.
- L is a nominal payments matrix, which is an n*n matrix, it includes the nominal liabilities between all nodes, where the value at the i_{th} row and j_{th} column represents L_{ij}. One restriction is that $\forall i, L_{ii} = 0$, the values on the diagonal of the matrix are also 0 since a firm would never owe itself a liability. Additionally, all other values in the matrix are non-negative, that is, $\forall i,\ j \in N, L_{ij} \geq 0$.
- e is an operating cash flow vector that includes the operating cash flow for each of the n nodes in the system, where each element e_i represents the total cash flow received by node i. While it is assumed that $e_i \geq 0$ in this model, this condition is actually not restrictive since negative cash flow can occur due to operating costs. It is important to note that operating costs should not be categorized as negative cash inflows, but rather represent all other liabilities to external factors of production (e.g., workers and suppliers). Therefore, the costs that exceed the revenues are not counted as negative cash balance, but instead as the liabilities to the workers. To address this, a "sink node" can be added to the financial system as node 0, which captures this concept. The sink node is assumed to have no operating cash flow (i.e., $e_0 = 0$) or obligations to other nodes (i.e., $L_{0j} = 0$ for all j). Moreover, the operating cost of each node i is treated as the liabilities of node i to the sink node 0 (i.e., L_{i0}). In this model, we allow such a sink node to exist, so that we can assume $e_i \geq 0$ without a loss of generality.
- p = (p_1, p_2, ..., p_n) is the total payment vector, where p_i represents the total payment made to all other nodes by node i.
- $\bar{p}$ = ($\bar{p}_1$, $\bar{p}_2$, ..., $\bar{p}_n$) is the total obligation vector, where $\bar{p}_i$ represents the total nominal obligation of node i to all other nodes, as given by [9]:

$$\bar{p} = \sum_{j=1}^{n} L_{ij}. \tag{3.1}$$

- Π is the relative liability matrix, where each element Π_{ij} represents the proportion of node i's liability to node j in the total liabilities of node i, as given by [9]:

$$\Pi_j \equiv \begin{cases} \frac{L_{ij}}{\bar{p}_i} & \text{if } \bar{p}_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

  In the present model, all debt claims are assumed to have equal priority, which means the payment made by node i to node j should be denoted as π_{ij}, while the total payments received by node i should be $\sum_{j=1}^{n} \Pi_{ij}^{T} p_j$. Moreover, as all the debtors and creditors are represented by individual nodes in this financial system,

it is expected that for any node i, its liability proportion to other nodes should sum to 1, as stipulated in [9]:

$$\forall i, \sum_{j=1}^{n} \Pi_{ij} = 1, \tag{3.3}$$

and in matrix form, as in [9]:

$$\Pi \mathbf{1} = \mathbf{1}. \tag{3.4}$$

- Finally, the clearing payment vector p^* is the output computed by the algorithms, which is the optimum clearing payment vector when each node pays most under three criteria. The element p_i^* represents the clearing payment of node i, in other words, the most total payment that node i should pay to all other nodes.

With all the definitions and notations mentioned above, the financial system can now be described by the triple (p, $\bar{p}$, e). Additionally, since the total cash flow received by a node i consists of both the payments received from other nodes and its operating cash flow (i.e., e_i), we can express the total cash flow to node i as follows:

$$\sum_{j=1}^{n} \Pi_{ij}^T p_j + e_i. \tag{3.5}$$

Furthermore, the determination of payment vector P should reflect the payments made by each node n in the financial system. These payments must adhere to the legal regulations mentioned earlier, i.e., limited liability, the priority of debt claims, and proportionality. Keeping these conditions in mind, and with all the notations that have been defined, the following definition can be made for the clearing payment vector P [9]:

Definition 3.1.1. The *clearing payment vector* for the financial system(Π, $\bar{p}$, e) is a vector $p^* \in [0, \bar{p}]$ that satisfies the following conditions:

a. *Limited Liability.* $\forall i \in N$,

$$p_i^* \leq \sum_{j=1}^{n} \Pi_{ij}^T p_j^* + e_i. \tag{3.6}$$

b. *Absolute Priority.* $\forall i \in N$, either obligations are paid in full, that is, $p_i^* = \bar{p}_i$, or all values are paid to creditors, that is,

$$p_i^* = \sum_{j=1}^{n} \Pi_{ij}^T p_j^* + e_i. \tag{3.7}$$

3.2 Algorithms in Theory

In this section, we will introduce and explain the theories and details of all three algorithms that will be implemented and experimented with in the following sections

using the financial model and notations defined previously. All three algorithms are used to compute the clearing payment vector in the financial system with only debt contracts. The first two algorithms are inspired by Eisenberg and Noe and the third algorithm, named the "Fictitious Default Algorithm," is explicitly introduced in [9].

Before delving into the algorithms, it is essential to determine whether clearing payment vectors exist in this system. In other words, it needs to be determined whether there is always a solution to the clearing problem algorithms. Fortunately, this has been proven by Eisenberg and Noe, and below is a summary of their proof [9].

3.2.1 Proof of the existence of clearing payment vectors

The following theorem, denoted as Theorem 1, is used to prove the existence of clearing payment vectors:

Theorem 1. For every financial system (Π, $\bar{p}$, e), there exists a greatest and least clearing payment vector, p^+ and p^-, respectively.

Before presenting the proof, a fixed-point characterization p_i of clearing vectors is introduced. As previously mentioned, clearing vectors must adhere to the principles of limited liability and absolute priority. Therefore, $p^* \in [\mathbf{0}, \bar{p}]$ is a clearing vector if and only if the following condition holds: $\forall i \in \mathcal{N}$:

$$p_i^* = \min[e_i + \sum_{j=1} \Pi_{ij}^T p_j^*, \bar{p}_i], \tag{3.8}$$

where $e_i + \sum_{j=1} \Pi_{ij}^T p_j^*$ represents "what the node i has" (i.e., the total value of node I, which is computed by the total cash flow of node i added to the total amount of money node i could receive from all other nodes under clearing vector p^*) and $\bar{p}_i$ represents "what the node i owns" (i.e., the total nominal liabilities of node i). Then, we can see that p^* is a fixed point of a map $\Phi(.;\Pi,\bar{p},e) : [\mathbf{0},\bar{p}] \to [\mathbf{0},\bar{p}]$ such that:

$$\Phi(p;\Pi,\bar{p},e) \equiv (\Pi^T p + e) \wedge \bar{p}. \tag{3.9}$$

The map can be easily understood as the nodes' value$\wedge$nominal obligation of nodes. We utilize the map Φ to complete the proof, where the set of fixed points of Φ is denoted as FIX(Φ). As discussed previously, Φ is composed of an affine map $q \to \Pi^T q + e$ and a positive, increasing, and concave map $q \to q \wedge \bar{p}$. Therefore, we know that Φ is also positive, increasing, and concave, and that $\Phi(\mathbf{0}) \geq \mathbf{0}$ and $\Phi(\bar{p}) \leq \bar{p}$.

According to Tarski's fixed-point theorem [19], as introduced in Section 2.3, FIX(Φ) is shown to be non-empty and forms a complete lattice, thereby establishing the existence of a greatest element p^+ and a least element p^-. Thus, the above theorem is proven, and the algorithms aim to find the greatest fixed point.

3.2.2 Algorithm - linear programming

The first algorithm employs the concept of linear programming. Linear programming, also known as linear optimization, is a mathematical method used to find a vector

$x^* \in \mathbb{R}^n$ that maximize or minimize the value of a linear objective function subject to certain constraints. In other words, it aims to find an optimal solution to a given problem. More specifically, a linear programming problem instance can be described in matrix and vector form as follows [17]:

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } A\mathbf{x} \leq \mathbf{b}.$$

Here, the three components of a linear programming problem are:

- A linear objective function: $\mathbf{c}^T\mathbf{x} = c_1x_1 + ... + c_nx_n$, where $\mathbf{c} \in \mathbb{R}^n$ is a given vector and c_i are rational numbers.
- Linear Constraints: $A\mathbf{x} < \mathbf{b}$ represents a set of linear inequalities, where A is a given m*n real matrix and $b \in \mathbb{R}^n$ is a given vector.
- An Optimization Criterion: Here, the criterion is maximization, but it could also be minimization. These can be freely converted since min f(x) = -max -f(x).

A feasible solution for a given linear program is any vector $\mathbf{x} \in \mathbb{R}^n$ that satisfies all of its constraints, while an optimal solution (optimum in short), is any $\mathbf{x}^* \in \mathbb{R}^n$ that yields the maximum possible value of $\mathbf{c}^T$ among all feasible $\mathbf{x}$.

According to [17], "[What] one should remember forever is this: A linear program is efficiently solvable, both in theory and in practice." The algorithms can solve each linear programming problem in polynomial time that is bounded by the input size. Therefore, with the definition of linear programming and the confidence of solving it efficiently, we can define our clearing payment problem as [9]:

$$\mathbf{p}(\Pi, \bar{p}, e, f) \quad \text{Max} f(p) \text{ where } p \in [0, \bar{p}]$$

$$s.t. \qquad p \leq \Pi^T p + e.$$

Here, we have a financial system $(\Pi, \bar{p}, e)$, and function f: $[0, \bar{p}] \to \mathbb{R}$, where each p is constrained between 0 and $\bar{p}$. The objective is to find the clearing vector that maximizes payments made by all nodes while adhering to the condition of limited liability, which is expressed by the constraint $p \leq \Pi^T p + e$. With this constraint and the optimization criterion, the remaining problem is to find the objective function f that ensures our solution to the linear programming problem is also the optimized solution to the clearing problem. To address this, Eisenberg and Noe prove the following lemma [9]:

Lemma 1. If f is increasing, then any solution to $\mathbf{P}(\Pi, \bar{p}, e, f)$ is a clearing vector for the financial system.

Therefore, Lemma1 implies that the clearing vector p^* can be found if we formalize f as a monotonic increasing function, that is, if $p \leq p'$, then $f(p) \leq f(p')$. To establish such a function is straight forward as we can simply write it as maximizing $\sum_{i=1}^n p_i$, so that when p increases, $\sum_{i=1}^n p_i$ also increases. We only need to ensure that the coefficients before each p_i are positive. Therefore, the simplest objective function is $f(p) = \sum_{i=1}^n p_i$.

3.2.3 Algorithm - Value Iteration

The second algorithm is fairly straightforward; it employs function iteration until convergence and utilizes the idea from the proof in Section 3.2.1, where we confirmed the existence of the maximum fixed point when using the map Φ. Since our aim is to maximize each value in p^*, and we know that $\Phi(\bar{p}) \leq \bar{p}$, the upper range can never exceed $\bar{p}$. If we start from the top and consistently apply the map Φ to the previous result, each value would gradually decrease as the mapping always takes the minimum between the value and the nominal obligation of the node. Eventually, the values will converge to the fixed point p^*, which is the optimal clearing payment vector. This idea can be illustrated as follows:

$$p^0 = \bar{p}; \qquad p^* = \lim_{i \to \infty} \Phi^i(p^0), \tag{3.10}$$

where Φ is the map introduced in Equation 3.9, and mapping is repeatedly applied until every value in vector p converges to a value.

The pseudocode for this algorithm is presented below in Algorithm 1. The input parameters are the matrix Π containing the fractions of the relative liabilities, the nominal liability vector $\bar{p}$, and the cash flow vector e. The output is the clearing vector, which is the optimum solution we characterized as p^* above.

Algorithm 1 The pseudocode for the algorithm in Section 3.2.3, inspired by [9].

Input: $\Pi, \bar{p}, e$
Output: p

$p \leftarrow \bar{p}$
$p_{old} \leftarrow$ random list of length n
while $p \neq p_{old}$ **do**
    $p_{old} \leftarrow p$
    $p \leftarrow (\Pi^T p + e) \wedge \bar{p}$
end while
return p

3.2.4 Algorithm - Fictitious Default Algorithm

The third algorithm, known as the "Fictitious Default Algorithm," simulates the default process of the nodes on a rolling basis instead of computing them all at once. It considers cascading defaults round by round. To provide a detailed illustration, in the first round, the algorithm assumes that all other nodes would be able to complete their obligations and calculates the total value of each node by summing up all the inflows each node should receive. If, despite this assumption, there are nodes that cannot complete their obligations (i.e., the total value of the node is less than the total liabilities of the node), those nodes are said to be in default. Otherwise, if all nodes can repay their liabilities, the algorithm terminates. In the second round, the algorithm again computes the values with the assumption that only a first-order default occurs and checks for any further defaults. Recall that a node is said to be in default once its total inflows cannot cover its

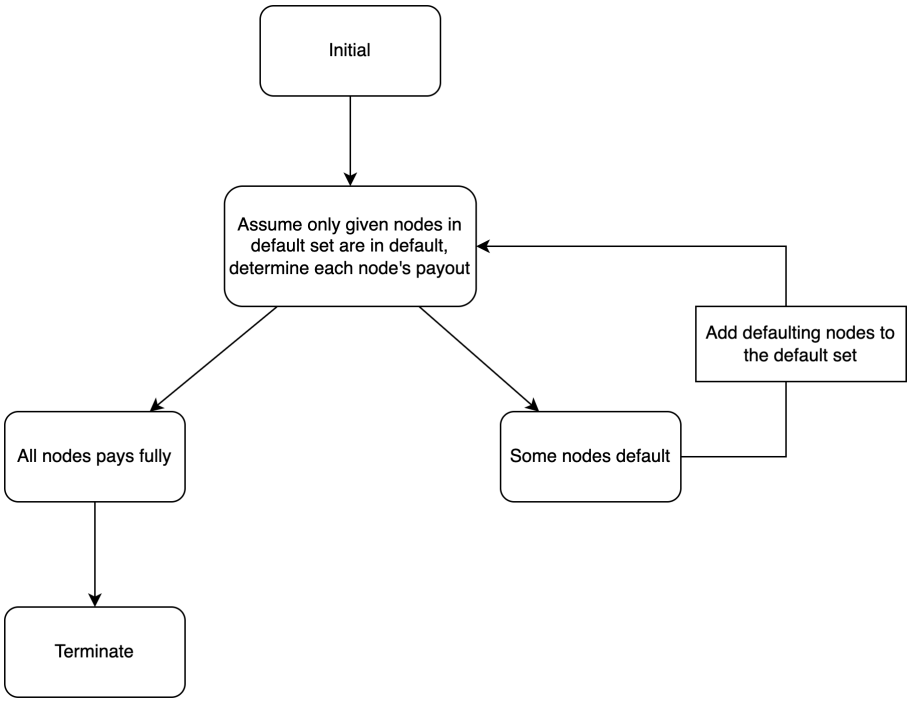

Figure 3.1: The workflow of the Fictitious Default Algorithm.

total liabilities. The algorithm continues to bring defaulting nodes to the third round and so on until no more nodes are in default, at which point the algorithm terminates. The workflow of the algorithm is illustrated in Figure 3.1.

As the algorithm eliminates at least one defaulting node per round, and there are n nodes in the financial system, the algorithm will stop after at most n rounds. An important economic advantage of this algorithm is its ability to track the order in which nodes default, providing a measure of each node's susceptibility to systemic risk in the clearing system. This order is partitioned into numbered rounds, ranging from 1 to n-1. For example, if node A defaults in the third round, we can infer that the financial health of node A is worse than node B, which defaulted in the fifth round. This information can be used to determine which nodes are at a higher risk of default and need to be closely monitored. However, the nodes that default in the first round are fundamentally insolvent, as they would default even without any exposure to systemic risk. These financial institutions need to take immediate measures.

With a high-level understanding of the algorithm, we will now delve into the details, including further explanations and analysis. However, before proceeding, it is necessary to introduce some additional notations and concepts.

- $\bar{S}$ is the set of supersolutions of the fixed-point operator Φ. In this context, supersolutions are the payment vectors in which some nodes pay more than their total inflow, according to the rules defined in Definition 3.1.1, i.e.,

$$\bar{S} = \{p \in [\mathbf{0}, \bar{p}] : \Phi(p) \leq p\}. \tag{3.11}$$

- p' is one of the supersolutions in $\bar{S}$, i.e., a fixed point $p' \in \bar{S}$.

- $\mathbf{D}(p)$ is the set of default nodes under the specific clearing vector $p \in \bar{S}$, such that $\Phi(p)_i < \bar{p}_i$.
- Λ is a diagonal n*n matrix under the specific clearing vector p, in which the diagonals equal 1 when the rows representing the nodes are in default, and equal 0 otherwise, that is:

$$\Lambda(p)_{ij} = \begin{cases} 1 & i = j \text{ and } i \in \mathbf{D}(p) \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

  Therefore, when multiplied with other matrices or vectors, this matrix only retains entries corresponding to defaulting nodes and sets the rest to zero. On the other hand, the complementary matrix $\mathbf{I} - \Lambda(p')$ would retain entries corresponding to non-defaulting nodes and sets the rest to zero.

Then, we can define a map $p \to FF_{p'}(p)$ for fixed p':

$$FF_{p'}(p) \equiv \Lambda(p')(\Pi^T(\Lambda(p')p + ((I - \Lambda(p'))\bar{p})) + e) + (I - \Lambda(p'))(\bar{p}), \tag{3.13}$$

where $\Lambda(p')$ is determined by the default nodes under p'. As discussed earlier, to maximize the clearing payment vector while adhering to the rules, we should assume that defaulting nodes pay all they have (i.e., the node's value), and non-defaulting nodes pay off all their nominal obligations. The map does this by using $\Lambda(p')$ to separate the defaulting and non-defaulting nodes and assigning them to the node's value or $\bar{p}_{node}$. Furthermore, Eisenberg and Noe proved that the map $\Lambda(p')$ has a unique fixed point, denoted as $f(p')$ [9]. Then, the sequence of payment vectors can be defined inductively as follows:

$$p^0 - \bar{p}; \qquad p^j = f(p^{j-1}). \tag{3.14}$$

Here, the sequence of vectors computed is referred to as the fictitious default sequence, while the process is called the Fictitious Default Algorithm. If the set of defaulting nodes remains unchanged at p^{j+1} and p, then p^j is the fixed point of Φ, and the sequence will remain constant after p^{j+1}. As mentioned earlier, the sequence will decrease to the clearing vector, which can be found in at most n iterations of the algorithm. This was proven by Eisenberg and Noe using induction [9]. Hence, this algorithm is computationally efficient and solvable in polynomial time.

The details of this algorithm are presented in pseudocode in Algorithm 2. The inputs and outputs are the same as in the previous algorithm. Note that the $Default\_set$ is the set D, which keeps a record of the defaulting nodes, and $isDefault$ is a Boolean value used to check if there are still nodes defaulting. The $\Lambda(p')$ matrix is always under clearing payment p, which means it updates in each round with the updates of the $Default\_set$. V_{nodes} represents the vector of the total value of all nodes. Notice that between lines 7 and 10, the while loop is attempting to find a fixed point of $f(p')$, but the fixed point can also be computed in other ways, which will be further discussed in the experiments section.

Algorithm 2 The pseudocode for the Fictitious Default Algorithm [9].

Input: $\Pi, \bar{p}, e$
Output: p

$p \leftarrow \bar{p}$
$p_{old} \leftarrow \bar{p}$
$Default\_set \leftarrow [\,]$
$isDefault \leftarrow True$
while $isDefault$ **do**
    Update $\Lambda(p') \leftarrow$ a n*n diagonal matrix, where the diagonal equals 1 when the node is in Default set, and equals 0 otherwise.
    while not $p_{old} =$p **do**
        $p_{old} \leftarrow p$
        $p \leftarrow \Lambda(p')(\Pi^T(\Lambda(p')p + ((I - \Lambda(p'))\bar{p})) + e) + (I - \Lambda(p'))(\bar{p})$
    end while
    $isDefault \leftarrow False$
    $V_{nodes} \leftarrow \Pi^T p + e$
    for idx, value in enumerate(V_{nodes}) **do**
        if idx not in Default_ set and (value $\leq p[idx]$) **then**
            $isDefault \leftarrow True$
            add idx to Default set
        end if
    end for
end while
return p

3.3 Implementation Details

In this section, we will discuss the implementation details of the algorithms introduced in the previous section, which is also the main contribution of this project. We will cover the functions and packages used to implement the algorithms and provide code snippets where it may be difficult to describe in words.

For the financial system with debt contracts only, all three algorithms have been implemented using Python version 3.8.3 on Jupyter Notebook version 6.0.3, which offers several benefits. Firstly, Python is easy to understand, with more intuitive syntax and fewer complicated class structures. Secondly, Python provides numerous well-documented packages and libraries, such as `NumPy` [12] and `Pandas` [23], that enable us to leverage existing code and functionality. This feature makes tasks such as matrix manipulation easier and more efficient, which is valuable since matrices and vectors play a crucial role in this project. In addition, Jupyter Notebook was used to write and test our algorithms in code blocks and to create markdowns for recording our progress and thoughts during implementation. This feature not only facilitated the amendment of our work but also enhanced the reproducibility of the results.

3.3.1 Implementing algorithm - linear programming

The first algorithm utilizes the idea of linear programming. As introduced in the theory section, we want to maximize $\sum_{i=1}^{n} p_i$ for each $p \in [0, \bar{p}]$ subject to the definition of limited liability. Therefore, the problem can be formalized as follows:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^{n} p_i \\ s.t. \quad & && p \leq \Pi^T p + e \\ & && p \leq \bar{p} \\ & && p \geq \mathbf{0} \end{aligned}$$

There are various Python libraries available for solving linear programming problems, and one commonly-used open-source library is SciPy [22]. It offers many fundamental algorithms for scientific computing, including those for solving linear programming. In this project, we utilized SciPy version 1.10.1 , and we specifically used the function `scipy.optimize.linprog`.

This function enables the user to choose a solver from a set of available solvers by specifying the 'method' parameter. The available solvers include HiGHS simplex [13] and interior-point method solvers, both of which have the same accuracy. However, the speed of the solver may depend on the specific problem, and this will be investigated further in the experiments.

The objective function of this function takes the form $min_x c^T x$, where x is the vector of decision variables. To convert it to the required form, we modify the objective function to $min_x -\mathbf{1}^T p$, where $\mathbf{1}$ is the vector of ones with length n since multiplying the vector

of ones by p gives the sum of the values in p. Since we are maximizing the result, we add a negative sign here.

The constraints need to be converted to the form $A_{ub}x \leq b_{ub}$, where A_{ub} is a matrix and b_{ub} is a vector. Therefore, we convert our constraint to $(\mathbb{I} - \Pi^T)p \leq e$, where $\mathbb{I}$ is the identity matrix. Additionally, the function allows us to have constraints of the form $l \leq x \leq u$, where l and u are vectors that bound each x. In our implementation, this would be represented as $\mathbf{0} \leq \bar{p}$.

3.3.2 Implementing algorithm - Value Iteration

The implementation of the Value Iteration Algorithm is straightforward. Here, for all the vectors and matrices, we use NumPy version 1.24.2 as it provides flexibility for manipulating arrays and offers numerous powerful mathematical functions. Additionally, arrays consume much less memory than data structures like lists. To begin, we initialize the clearing vector as $\bar{p}$, and for comparison purposes, also initialize a vector p_{old} with random values. The vectors then enter a while loop that continues until p is equal to p_{old}. At each iteration, p is updated as $(\Pi^T p + e) \wedge \bar{p}$ and p_{old} is set to be equal to the previous value of p. The only part that could affect the speed of the entire iteration process is the implementation of the map Φ. During implementation, we utilized the function `numpy.minimum`, which performs element-wise comparisons between two arrays and returns a new array containing the minimum values from the input arrays.

3.3.3 Implementing algorithm - Fictitious Default Algorithm

To implement the Fictitious Default algorithm, we follow the pseudocode presented in Algorithm 2. The key idea of this algorithm is to simulate the nodes going into default round by round. To keep track of the defaulting nodes, we update a set called $Default\_set$ and use a Boolean variable called $isDefault$ to check if any nodes are still defaulting, which determines whether the algorithm should stop. It is important to update the diagonal matrix Λ according to $Default\_set$ for further mapping, that is, set the diagonal elements to 1 when the corresponding node is in the $Default\_set$. For example, if node 2 is in the $Default\_set$, then we should also set $\Lambda[1][1]$ to 1. To do this, we first create a vector of length n that has a value of 1 when its index corresponds to the index stored in the $Default\_set$. Then, we use the function `numpy.diag` with the vector as input to construct a diagonal array with vector[i] on the i_{th} diagonal. The code snippet for this part is shown below:

```
D_diag = np.zeros(N)
for i in Default_set:
    D_diag[i] = 1
lambda_diag = np.diag(D_diag)
```

Listing 3.1: Code snippet for updating the diagonal matrix λ at each round according to the Default_set for N nodes.

Next, the algorithm proceeds to the mapping step using the FF map introduced in Equation 3.13. All multiplications between matrices and vectors should be performed

using `numpy.dot`, as they involve either matrix-matrix multiplication or vector-matrix multiplication. One challenge in this algorithm is finding the fixed point p' at each round. Several approaches can be employed to find the fixed point. As demonstrated in the pseudocode, we can continuously apply the FF map until the value of p is unchanged. Alternatively, other techniques such as repeatedly applying the Φ map can also be applied to find the fixed point. We will explore and compare the most effective techniques in next section.

Lastly, before moving to the next iteration, we update the default set by calculating the nodes' values and comparing them with their total nominal liabilities. If there are nodes that cannot cover their liabilities and are not yet in the default set, their indices are added, and the Boolean value is updated to continue iterating. The nodes' values are calculated using $\Pi^T p + e$, where p is the fixed point p' that was just determined.

3.4 Experiment

In this section, we will conduct experiments with different versions of algorithm implementations to identify the optimal approach. Furthermore, we will simulate a real financial network by initializing with reasonable data and testing the algorithms implemented so far to evaluate their performance. We will compare all three algorithms using the same data to assess their effectiveness in terms of speed and accuracy. Through the designed experimentation, we gain insights into the strengths and weaknesses of each algorithm, as well as their usefulness in real-world applications.

3.4.1 Data initialization

Initializing the data to resemble a real financial network is crucial when assessing systemic risk in financial systems. In our experiments, we assume that all financial institutions are banks, and their liabilities are interbank loans. In the financial model by Eisenberg and Noe, to perform experiments and find the clearing vector, the structure of bilateral nominal liability must be provided. As previously defined, we represent this as an n*n nominal liability matrix L. Thus, the structure of L with initialized data that follows the rules specified in Section 3.1 should be:

$$L = \begin{bmatrix} 0 & \ell_{12} & \dots & \ell_{1n} \\ \ell_{21} & 0 & \dots & \ell_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & \ell_{n2} & \dots & 0 \end{bmatrix}$$

Finding appropriate data to fill the matrix L is no easy task. While several studies have conducted simulations to assess the risk of contagion arising from exposures in the interbank loan market, bilateral loan data is often scarce and of limited quality since banks are not required to disclose their counterparties [21]. Consequently, this information is unobservable to anyone except the involved parties, and access to electronic trading platforms is available only in a limited number of jurisdictions [20]. Alternatively, some literature estimates data from balance sheets or payment

data, such as [8], which simulates using data from a confidential database that includes banks' balance sheet statements and other key financial figures. Inference techniques, such as Maximum Entropy [21] and the RAS algorithm [5], are used to estimate the incomplete parts. However, it has been argued that the entropy method is unable to replicate certain stylized characteristics of interbank markets, including the sparseness of the interbank liability relationship matrix or the presence of tiering [20]. For example, [6] demonstrated that banks only engage in transactions with a limited subset of other banks, while [21] provided evidence supporting the existence of tiering.

Some studies, such as [7], use data from a comprehensive set of banking statistics on large loans and concentrated exposures compiled by the Deutsche Bundesbank without employing any entropy methods to maintain the network structure. However, accessing data from Deutsche Bundesbank [1] requires feasibility checks. Since our project focuses more on the effectiveness of the algorithms, generating experimental ourselves that aligns with the features of the financial network is preferred. This approach also provides greater flexibility to adjust parameters, such as the number of banks in the network, to better test the algorithm. The key consideration is to carefully account for the characteristics of the financial network and strive to generate more realistic data.

To capture the characteristics of the financial network, such as the tiering mentioned earlier, various studies have identified that financial networks for interbank loans exhibit a core-periphery structure. In [7], for instance, the market is not illustrated as a centralized exchange, but rather a sparse network centred around a group of core banks that act as intermediaries (i.e., a bank acting both as lender and borrower) between numerous smaller banks in the periphery. The banks are partitioned according to their bilateral relations, as listed below [7]:

(1) Top-tier banks lend to each other

(2) Lower-tier banks do not lend to each other

(3) Top-tier banks lend to lower-tier banks

(4) Top-tier banks borrow from lower-tier banks

Here, the top-tier banks are called the core, which are explained as special intermediaries that are important in holding together the interbank market, and the lower-tier banks are called the periphery. A simple example of an interbank market with a core-periphery model structure is presented in Figure 3.2.

We implement the core-periphery structure during matrix initialization to simulate the real characteristics of the financial network. To create an $n \times n$ empty matrix, we use the function `numpy.zeros`. Next, we populate the matrix according to the following rules: If both nodes are in the core, a connection is made with probability 1 (as per Relation 1); if one node is in the core and the other is in the periphery, a connection is made with a specified probability (as per Relations 2 and 3); if both nodes are in the periphery, a connection is not made (as per Relation 4). It is important to note that the core nodes are ordered first in the matrix. An example code snippet for generating random values when one node is in the core and the other is in the periphery is shown below. During the current iteration, `row` and `col` represent the row and column numbers, respectively.

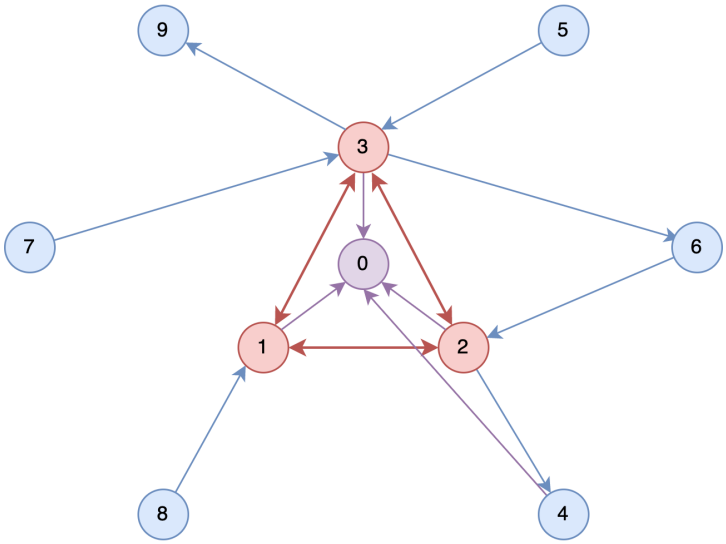

Figure 3.2: A simple example of an interbank market with a core-periphery structure. The arrows indicate the direction of payment flow, for instance, node 5 has a promised payment to node 3 (i.e., L_{53} has a value). In this example, node 0 is characterized as the sink node as mentioned in Section 3.1, so only L_{i0} exists. Nodes 1, 2, and 3 are part of the core, while the other nodes belong to the periphery.

We use the function `numpy.random.rand` to generate a random probability and the function `numpy.random.randint` to generate a random liability value to fill in the matrix. In this case, the `core_connection_prob` is set to 1.

```
if row < num_core and col < num_core:  # Both nodes are the core
    if np.random.rand() < core_connection_prob:
        matrix[row, col] = np.random.randint(1,100)
```

Listing 3.2: Code snippet for populating the core-periphery matrix with random values when both nodes are in the core.

Afterward, to capture the characteristic of the sink node in this financial network, we set $L_{0i} = 0$ for all i and randomly set a proportion of the L_{i0} to 0. To ensure reproducibility, we use the function `numpy.random.seed` to control the randomness of the generated data. An example of a generated matrix with 6 nodes is shown below, where we have a sink node, 3 core nodes, and 2 periphery nodes. The probability of having a relation between a core node and a periphery node is set to be 0.6, the proportion of dropping the L_{i0} values is 0.2, and the random seed 4 is used.

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 37 & 0 & 39 & 53 & 0 & 22 \\ 0 & 31 & 0 & 67 & 33 & 0 \\ 34 & 29 & 80 & 0 & 0 & 0 \\ 0 & 90 & 49 & 95 & 0 & 0 \\ 76 & 71 & 0 & 74 & 0 & 0 \end{bmatrix}$$

Finally, vector e can be easily generated using the function `numpy.random.randint`

according to the definition given in Section 3.1.

3.4.2 Comparison between different solvers in Algorithm using linear programming

Motivation: As mentioned in Section 3.3.1, we conduct experiments on the different solvers available in the `scipy.optimize.linprog` function, including the HiGHS simplex solver and the interior-point method solver. Since the speed of a solver depends on the specific problem [22], we aim to investigate which solver is best for solving this clearing problem.

Experiment settings: The liability matrix L and cash flow vector e are initialized with a number of nodes of 500, 1000, and 2000, respectively. The parameter `method` in the function `scipy.optimize.linprog` is changed, where `highs-ds` represents the HiGHS simplex solver and `highs-ipm` represents the interior-point method solver. To obtain the computation time, we used the module `time` in Python to record the start time and end time and calculate the elapsed time.(Same for the following experiments).

Results: The findings indicate that the elapsed time for the HiGHS simplex solver is consistently slightly shorter than the interior-point method solver, while maintaining the same level of accuracy. For comparison purposes, the sum of the values in the clearing payment vector is presented, since the vector itself is too long to be included in the report. However, the clearing vectors were also compared through observation. Nevertheless, since the results were very close, summing up the values in the clearing payment vectors provided a more accurate and intuitive way to compare. Therefore, the following experiments will also compare the summation of the clearing payment vectors. Here, we can see that the sum is exactly the same using two solvers, but the difference in elapsed time is approximately 4 seconds when the number of nodes reaches 2000. Based on these results, we will use the HiGHS simplex solver for subsequent experiments, as it gives a higher speed for solving the clearing problem. The results are summarized in Table 3.1 below:

No. of Nodes (n)	Solver	Elapsed time(s)	$\sum_{i=1}^{n} p_i^*$
500	HiGHS simplex	0.6752	5994568.830941179
500	interior-point method	0.8653	5994568.830941179
1000	HiGHS simplex	5.9766	24977078.073851094
1000	interior-point method	7.4894	24977078.073851094
2000	HiGHS simplex	58.0354	101647450.1118128
2000	interior-point method	62.3459	101647450.1118128

Table 3.1: Comparison between two solvers as the linear programming algorithm.

3.4.3 Fictitious Default Algorithm as the most effective way to find the fixed point

Motivation: As mentioned in Section 3.3.3, we aim to identify the most effective way to find the fixed point in the Fictitious Default Algorithm and analyse it in terms of the

speed and accuracy.

Experiment settings: We propose two ways to implement this part to find the fixed point. One is to use the Φ map in the second algorithm, while the other is to iterate the FF map until it converges. To highlight the differences, we initialize the number of nodes n to be 2000.

Results: It is clear that using the map Φ results in a significantly lower elapsed time while maintaining the same level of accuracy. Consequently, further experiments will also use the implementation version that employs the map Φ to find the fixed point. The results of the sum of the values in clearing vectors and elapsed time are summarized in Table 3.2 below:

	Φ map	FF map
Elapsed Time(s)	1.4020	80.8759
$\sum_{i=1}^{n} p_i^*$	101647450.1118128	101647450.1118128

Table 3.2: Comparison between the two ways to implement the Fictitious Default Algorithm.

3.4.4 Comparison of the three algorithms

Motivation: Since all three algorithms are used to find the clearing vector, this experiment aims to compare their performance and determine their respective strengths and weaknesses in different scenarios. Specifically, we aim to identify which algorithm is preferable under certain conditions.

Experiment settings: In this experiment, we will test the three algorithms by finding the clearing vector with varying numbers of nodes n. Specifically, we test with 100, 500, 1000, and 2000 nodes, with half of the nodes designated as core nodes. We use a random seed of 40 for vector e and 4 for the liability matrix L to generate the data.

Results: Figure 3.3 compares the elapsed time in finding the clearing vector as the number of nodes increases for the three algorithms. The elapsed time for Algorithm 2 (Value Iteration) and Algorithm 3 (Fictitious Default Algorithm) are quite similar for all numbers of nodes, with Algorithm 2 being slightly faster than Algorithm 3. However, the elapsed time of Algorithm 1 with a small number of nodes does not differ much from the other two algorithms. As the number of nodes increases, the elapsed time appears to increase non-linearly and is significantly steeper than Algorithms 2 and 3. For example, when the number of nodes is 1000, it takes Algorithm 1 less than 8 seconds to compute, while it takes nearly 60 seconds when the number of nodes increases to 2000.

Next, we compare the accuracy of the three algorithms. Recall that algorithms such as the Simplex method (which is the algorithm used in our implementation) in linear programming problems can always find the optimum solution. Comparing the results, we can see that the outcomes for Algorithm 2 and Algorithm 3 are always the same, while Algorithm 1 consistently provides a slightly better result, except when the number of nodes is 1000.

Indeed, each of the algorithms has its own advantages. Based on the results and the trade-off between speed and accuracy, Algorithm 1 can be used to obtain the optimum result when the number of nodes is small, taking advantage of its higher accuracy in those scenarios. However, when the number of nodes is significantly large, Algorithm 2 (Value Iteration) would be a better choice as it provides a good balance between computational efficiency and accuracy.

On the other hand, Algorithm 3 (the Fictitious Default Algorithm) takes slightly longer due to the additional computations it requires. Nevertheless, it has the advantage of outputting the round of defaulting nodes, which can be valuable when analysing the risk associated with individual nodes in the financial network. Thus, Algorithm 3 can be used when the focus is on risk analysis.

The results are also summarized below in Table 3.3:

Number of Nodes (n)	Algorithm	Elapsed time (s)	$\sum_{i=1}^{n} p_i^*$
100	1	0.0451	227696.01731738096
100	2	0.0052	227696.0173173804
100	3	0.0066	227696.0173173804
200	1	0.0559	942638.7075650934
200	2	0.0095	942638.7075650926
200	3	0.0102	942638.7075650926
500	1	0.8712	5994568.830941179
500	2	0.0567	5994568.830941178
500	3	0.0495	5994568.830941178
1000	1	7.8546	24977078.073851094
1000	2	0.2199	24977078.073851097
1000	3	0.3325	24977078.073851097
2000	1	59.8180	101647450.1118128
2000	2	1.2963	101647450.11181268
2000	3	1.4134	101647450.11181268

Table 3.3: Comparison between the three algorithms used to determine the clearing payment vector with varying number of nodes. Algorithm 1 refers to the linear programming method, Algorithm 2 refers to Value Iteration using the map Φ, and Algorithm 3 refers to the Fictitious Default Algorithm. The sum of the values of the clearing vector solutions is presented to show the comparison between the algorithms.

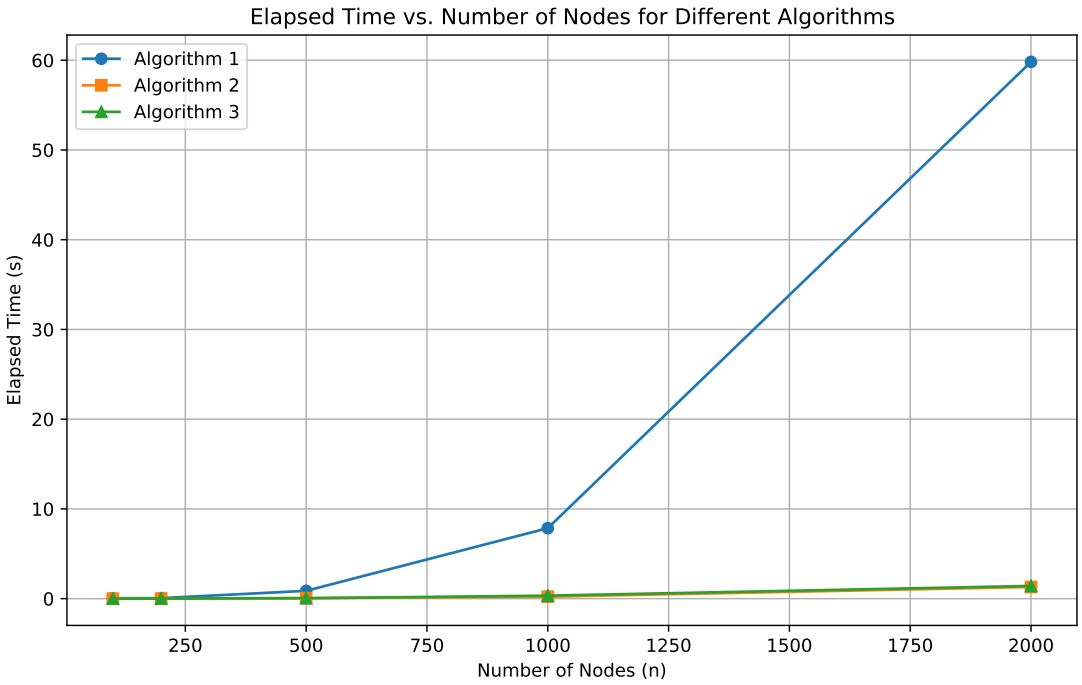

Figure 3.3: Comparison of elapsed time in seconds for different algorithms with varying number of nodes (n). Algorithm 1 refers to the linear programming method, Algorithm 2 refers to Value Iteration using the map Φ, and Algorithm 3 refers to the Fictitious Default Algorithm.

Chapter 4

Financial model with both debt and equity contracts

For further development, we aim to include additional contracts in the financial network and investigate the algorithm on this more complicated financial system. Jackson *et al.* extended previous models used to assess systemic risk, which were based solely on debt dependencies [9] or equity-like dependencies [10], to include both debt and equity contracts, as well as bankruptcies and discontinuous costs due to defaults, as these factors are crucial in causing inefficiencies and externalities in financial networks. In this chapter, we introduce this financial model as well as the algorithm they introduced to find the equilibrium banks' value, which can be used to calculate the clearing vector. Afterwards, the implementation details and a simple experiment are discussed.

4.1 The Financial Model of Jackson *et al.*

A variety of notations are used to formalize the financial system. It is worth emphasizing that, to maintain consistency with the original literature [14], some notations in this section use the same letter to represent different entities in the previous model.

Consider a financial network with n nodes, each representing a distinct economic entity or financial node. These nodes could have not only nominal liabilities but also equity contracts with other nodes in the system:

- N = $\{0, 1, ... , n\}$ is a set of financial organizations, such as banks (for brevity, we will refer to these organizations as 'banks' in the following). Node 0 represents all other actors who have contracts with these organizations but are not part of any organization. That is, node 0 could be private investors that hold equity in these organizations, or private companies that borrow money from organizations like banks. Node 0 is a crucial component in this financial network as it enables the modelling of interactions between financial organizations and external actors.
- K = $\{1, ..., K\}$ is the set of primitive assets, where p_k represents the present value (i.e., market price) of asset $k \in K$, so that $\mathbf{p} = (p_1,...p_k)$ is the vector of present values for k primitive assets. The organizations, banks for instance, construct

their portfolios by investing in these primitive assets externally, in addition to participating in financial contracts within the network. The primitive assets could be stocks or bonds and are therefore essential to present in the network.

- Matrix **q** then represents the quantities invested by each bank in each asset, where q[i][k] corresponds to the amount invested by bank i in asset k, denoted as q_{ik}. It is required that $q_{ik} \geq 0$. Therefore, the total value of primitive assets of a node i is $\sum_k q_{ik} p_k$.
- D_{ij} represents the face value (promised payment) of j's liability towards i, that is, node j is the debtor (the one who received money) and node i is the creditor. Note that as a bank cannot have debt claim on itself, $D_{ii} = 0$.
- **D** is the matrix of debt claims, which compromises all the values of debt contracts in between the nodes, so that D[i][j] represents D_{ij}. Then, the total nominal debt assets of a node i is denoted by $D_i^A \equiv \sum_j D_{ij}$, and the total nominal liabilities of a node i is denoted by $D_i^L \equiv \sum_j D_{ji}$.
- S_{ij} represents the equity claim of node i on node j's value, meaning that node i owns an equity share in node j. Therefore, S_{ij} is a fraction between 0 and 1 ($S_{ij} \in [0,1]$), representing the percentage of node j's value that is owned by node i. In our model, we assume that fully privately held banks have an external investor who owns an equity share of 1. This assumption does not affect the results, but it allows us to track the accumulation of values more easily. As a consequence, a bank cannot have an equity claim on itself, which means that $S_{ii} = 0$. Additionally, equity shares should sum to one (i.e., $\sum_i S_{ij} = 1$), indicating that any share not owned by other banks is attributed to an external investor. Thus, we have $S_{0j} = 1 - \sum_{i \neq 0} S_{ij}$. Then, no shares are held by outside investors so that $\forall i, S_{i0} = 0$, shares held by banks in private enterprises are modelled via the p_i vectors. For a well-structured economy, we need to eliminate unreasonable cycles in which banks solely own one another (e.g., $S_{23} = 1$ should not exist), without any private investor holding partial ownership. We do this by assuming a directed equity path that connects every bank to a private investor, that is, S_{0i} always has a value except for when i = 0.
- **S** is the matrix of equity claims, which compromises all the proportions of equity contracts in between the nodes such that S[i][j] represents S_{ij}.

We have defined the primitive asset, liability matrix, and equity matrix, however, Jackson *et al.* also considers the existence of the bankruptcy cost [14]. These costs represent the real-world costs incurred in financial systems when a bank defaults. The actual cost depends on the value of the bank or, more specifically, on the bank's equilibrium value, which is denoted by V_i. As these concepts are highly interrelated, they will be introduced together in this section with references made to one another to aid in understanding.

- First, the equilibrium banks' value vector is $\mathbf{V} = (V_0, V_1 ..., V_n)$. The equilibrium value refers to the fixed point of the banks' value, and each value $\mathbf{V}_i$ follows the Equation 4.1. The terms $d_j^A(\mathbf{V})$ and $b_i(\mathbf{V}, \mathbf{p})$ are introduced later in Equations 4.6

and 4.3, respectively.

$$V_i = \sum_k q_{ik} p_k + \sum_j S_{ij} V_j^+ + d_j^A(\mathbf{V}) - D_i^L - b_i(\mathbf{V}, \mathbf{p}) \tag{4.1}$$

- $\beta(\mathbf{V},\mathbf{p})$ is then the bankruptcy cost that a bank incurs when it is in default given the banks' value vector $\mathbf{V}$. The bankruptcy cost is computed by Equation 4.2 [14]:

$$\beta_i(\mathbf{V}, \mathbf{p}) = b + a \left[\sum_k q_{ik} p_k + \sum_j S_{ij} V_j^+ + d_i^A(\mathbf{V}) \right] \quad \text{with a} \in [0,1], \text{ b} \geq 0. \tag{4.2}$$

  In this context, b represents an additional fixed cost, while a is a fraction used to discount the bank's total assets during a default. Moreover, apart from $\sum_k q_{ik} p_k$, which signifies the total value of primitive investments, it is important to recognize that the default cost is also influenced by $d_i^A(\mathbf{V})$, representing the total value node i could receive under the bank value $\mathbf{V}$, which is interrelated to the bankruptcy cost and therefore is introduced below. $\sum_j S_{ij} V_j^+$, denoting the total value of equity shares owned by the bank under the bank values $\mathbf{V}$. Recall that $V_j^+ \equiv max\{V_j, 0\}$, since the equity should always be positive. As a result, the equation suggests that the default cost is affected by both the financial well-being of other banks and the value of the bank's primitive assets.

- Then, for

$$\mathbf{d}^A(\mathbf{V}) = \{d_1^A(\mathbf{V}), d_2^A(\mathbf{V}) ..., d_N^A(\mathbf{V})\} \tag{4.3}$$

  we define $d_i^A(\mathbf{V}) \equiv \sum_j d_{ij}(\mathbf{V})$, where $d_{ij}(\mathbf{V})$ is given by:

  - If j is solvent, it must pay all of its debts:

$$d_{ij}(\mathbf{V}) = D_{ij} \tag{4.4}$$

  - If j is insolvent, it should pay out all of its assets to its creditors, according to the proportion to their claim on j:

$$d_{ij}(\mathbf{V}) = \frac{D_{ij}}{\sum_h D_{hj}} max(\sum_k q_{jk} p_k + d_j^A(\mathbf{V}) + \sum_h S_{jh} V_h^+ - \beta_j(\mathbf{V}, \mathbf{p}), 0) \tag{4.5}$$

    where the $V_h^+ \equiv max\{V_h, 0\}$, the first term represents the fraction of i's claim to the total claim on j.

- Finally, $b_i(\mathbf{V}, \mathbf{p})$ represents the bankruptcy costs node i would incur, that is,

$$b_i(\mathbf{V}, \mathbf{p}) = \begin{cases} 0 & \text{if } \sum_k q_{ik} p_k + \sum_j S_{ij} V_j^+ + d_i^A(\mathbf{V}) \geq D_i^L \\ \beta_i(\mathbf{V}, \mathbf{p}) & \text{if } \sum_k q_{ik} p_k + \sum_j S_{ij} V_j^+ + d_i^A(\mathbf{V}) < D_i^L. \end{cases} \tag{4.6}$$

  Since the bankruptcy cost reduces the bank's value, as demonstrated in Equation 4.1, V_i can become negative when the bankruptcy cost exceeds the total assets owned by the bank.

All the definitions and notations used to formalize this financial network have been introduced. To better illustrate these concepts, we present an example of a simple relationship graph in Figure 4.1, consisting of only 3 banks. To determine the equilibrium banks' values, we calculate V_0, V_1, and V_2 separately, and then combine them to form the equilibrium banks' vector V.

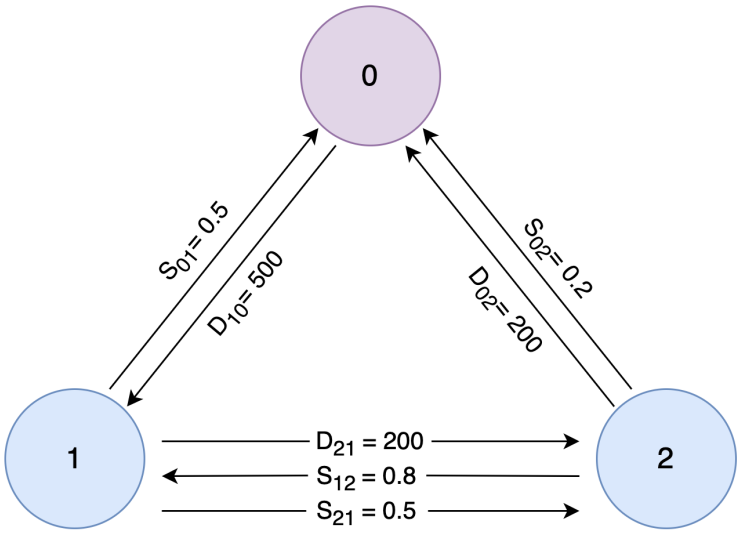

Figure 4.1: A simple financial network example with both debt and equity contracts, using the notation defined in Section 4.1. Here, we have Bank 0, Bank 1, and Bank 2. The arrows indicate the directions of payment flow. Therefore, Bank 0 borrowed from Bank 1 a value of 500, Bank 1 borrowed from Bank 2 a value of 200, Bank 2 borrowed from Bank 0 a value of 200; Bank 0 owns 50% of the shares of Bank 1, while the remaining 50% is owned by Bank 2. Bank 0 also owns 20% of the shares of Bank 2, while the remaining 80% is owned by Bank 1.

4.2 Algorithms in Theory

The theories, proofs, and pseudocode of the algorithm introduced in [14] will be clearly presented in this section.

Although Jackson *et al.* did not explicitly attempt to determine the clearing vector, they introduced the concept of a "bank's equilibrium value," which refers to the fixed point of Equation 4.7. By solving the following equation, we can determine the equilibrium value of each bank, which in turn makes finding the clearing vector a simple task.

$$\mathbf{V} = (\mathbf{I} - \mathbf{S}(\mathbf{V}))^{-1}([\mathbf{qp} + \mathbf{d}^A(\mathbf{V}) - \mathbf{D}^L] - \mathbf{b}(\mathbf{V}, \mathbf{p})) \tag{4.7}$$

Here, $\mathbf{V} = \{V_1, V_2 ..., V_N\}$ represents the vector of all V_i where each V_i is presented in Equation 4.1. $\mathbf{I}$ is the n*n identity matrix and $\mathbf{S}(\mathbf{V})$ denotes the equity relationship matrix under equilibrium vector $\mathbf{V}$. The superscript $'()^{-1}'$ here means the inverse of a matrix, consequently multiplying $(\mathbf{I} - \mathbf{S}(\mathbf{V}))^{-1}$ would add the equity value under $\mathbf{V}$ to every bank. $\mathbf{qp}$ is the vector that consists of the total value of owned primitive assets of each bank. $\mathbf{d}^A$ is the vector that consists of the total value that each bank would receive from other banks under $\mathbf{V}$. $\mathbf{D}^L$ is the vector of total obligations and $\mathbf{b}(\mathbf{V}, \mathbf{p})$ is the default cost for all banks under $\mathbf{V}$. Therefore, the above equation can be seen as the sum of the primitive asset and the inflow due to debt contracts, which is then subtracted from the total obligations and the bankruptcy cost. Finally, the equity value that the bank should receive under $\mathbf{V}$ is added to the equation.

Certain terms require a more detailed explanation and need more careful consideration. For $\mathbf{S}(\mathbf{V})$, we need to update $S_{ij}(\mathbf{V}) = 0$ whenever node j defaults, and S_{ij} otherwise, to reflect the fact that a bank should always repay its debt before equity. In other words, if a bank is insolvent, the equity holders of the bank should receive nothing from it.

The last term represents the bankruptcy cost that the bank incurs as given by Equation 4.6. This cost also changes with the value of the bank's assets.

4.2.1 Proof of the existence of the equilibrium value

At first, a bank's value depends monotonically on each other in the financial network, which means that as the values of other banks increase, the value of a bank in the system also weakly increases [14].

This can be shown by analysing the terms in Equation 4.1, excluding the constants $\sum_k q_{ik} p_k$ and D_i^L. The term $\sum_j S_{ij} V_j^+$ is non-decreasing as the equity value that bank i could receive increases when the values of other banks increase. The non-decreasing term of $d_j^A(\mathbf{V}) - b_i(\mathbf{V}, \mathbf{p})$ with respect to $\mathbf{V}$ can also be established. If bank i is solvent, an increase in $d_j^A(\mathbf{V})$ due to the increase in other banks' values would result in an increase in that term since the bankruptcy cost $b_i(\mathbf{V}, \mathbf{p})$ would be zero. If bank i is insolvent, the increased value of bankruptcy cost would be compensated since, according to Equation 4.2, the bankruptcy cost is a fixed cost plus a fraction of the total asset value in which the term $\sum_k q_{ik} p_k$ is deterministic. Thus, only the term $\sum_j S_{ij} V_j^+ + d_j^A(\mathbf{V})$ would change, but since a is always smaller than 1, the increased value is always a fraction of the term $d_j^A(\mathbf{V})$ according to Equation 4.5.

Next, it is shown that the banks' values are bounded by the maximum values of the banks' assets, denoted by $\bar{\mathbf{V}} = (\mathbf{I} - \mathbf{S}(\mathbf{V}))^{-1}[\mathbf{qp} + \mathbf{D}^A]$. The minimum value is $\mathbf{V} = \mathbf{qp} - \mathbf{D}^L - \mathbf{b}(\mathbf{V}, \mathbf{p})$, where $\mathbf{b}(\mathbf{V}, \mathbf{p})$ is the bankruptcy cost incurred by each bank.

Therefore, according to Tarski's fixed point theorem, it is shown that the set of fixed points of $\mathbf{V}$ is non-empty and forms a complete lattice, indicating that there always exists a maximum equilibrium value and a minimum. Therefore, the existence of the equilibrium value is proven.

4.2.2 Algorithm for equilibrium banks' value

Although multiple equilibrium values $\mathbf{V}$ exist in the system, our aim is to find the optimal one that maximizes the value of each bank. However, recalling Equation 4.7, we can see that this problem is different from the clearing vector problem solved before, as nonlinearities exist in the equation and therefore linear programming methods do not work here. For example, $\mathbf{d}^A(\mathbf{V})$ (mentioned in Equation 4.3) includes two parts of a calculation. As stated in Equation 4.4 and Equation 4.5, the value bank i can receive from j depends on the situation of bank j, and when bank j is insolvent, computation of a maximum ($max\{V_h, 0\}$) is necessary to ensure that only a positive value is paid to bank i.

Inspired by Section 3.1 in [15], which gives a simple example of computing the bank values, we separate the solvent and insolvent banks and solve the equilibrium value by

computing the best equilibrium value starting from the upper bound, that is:

$$\mathbf{V}^0 = (\mathbf{I} - \mathbf{S}(\mathbf{V}))^{-1}[\mathbf{qp} + \mathbf{D}^A] \tag{4.8}$$

Subsequently, we update and compute the banks' values. For example, in the next round, we would compute:

$$\mathbf{V}^1 = (\mathbf{I} - \mathbf{S}(\mathbf{V}^0))^{-1}([\mathbf{qp} + \mathbf{d}^A(\mathbf{V}^0) - \mathbf{D}^L] - \mathbf{b}(\mathbf{V}^0, \mathbf{p})). \tag{4.9}$$

Continuously computing the equilibrium value in this way will eventually yield the best equilibrium value. However, when computing each round, a problem arises in updating the matrix $\mathbf{S}$ so that $S_{ij}(\mathbf{V}) = 0$ when j defaults, since it affects every term in the equation. To address this issue, the Fictitious Default Algorithm [9] can be employed. This algorithm keeps track of the default banks, allowing us to update the matrix $\mathbf{S}$ accordingly. Moreover, this approach simplifies the calculation of $d_{ij}(\mathbf{V})$ and $b_i(\mathbf{V}, \mathbf{p})$ when we need to consider their computation in two different situations, i.e., if bank j is solvent or insolvent.

The details of this algorithm are formalized as pseudocode in Algorithm 3. Here, $Default\_set$ keeps track of the default banks, and $isDefault$ is a Boolean value used to check if there are still banks defaulting. The input of the algorithm includes the liability matrix $\mathbf{D}$, the equity matrix $\mathbf{S}$, and the vectors $\mathbf{p}$ and $\mathbf{q}$ for the primitive assets of banks. The output is the vector $\mathbf{V}$, which represents the equilibrium banks' value.

Algorithm 3 Algorithm for finding the banks' equilibrium value.

Input: $\mathbf{S}, \mathbf{D}, \mathbf{p}, \mathbf{q}$
Output: $\mathbf{V}$

$\mathbf{V}^0 \leftarrow (\mathbf{I} - \mathbf{S}^0)^{-1}(\mathbf{qp} + \mathbf{D}^A)$ {$\mathbf{D}^A$ is a vector of D_i^A, where $D_i^A \equiv \sum_j D_{ij}$}
$Default\_set \leftarrow [\,]$
$isDefault \leftarrow True$
while $isDefault$ **do**
  $isDefault \leftarrow False$
  $\mathbf{V} \leftarrow (\mathbf{I} - \mathbf{S}(\mathbf{V}))^{-1}([\mathbf{qp} + d^A(\mathbf{V}) - \mathbf{D}^L] - b(\mathbf{V}, \mathbf{p}))$
  $all\_default\_index \leftarrow$ index where $(\mathbf{I} - \mathbf{S}(\mathbf{V}))^{-1}[\mathbf{qp} + \mathbf{D}^A] < \mathbf{D}^L$
  if $0 <$ len($all\_default\_index$) $<$ n-1 **then**
    $S_{ij} \leftarrow 0$ for j in $all\_default\_index$
    for index in $all\_default\_index$ **do**
      if index not in $Default\_set$ **then**
        Add index to $Default\_set$
        $isDefault \leftarrow True$
      end if
    end for
  end if
end while
return $\mathbf{V}$

Finally, once the best equilibrium bank values are computed, we can easily compute the clearing vector by calculating $\sum_{i=0}^{N} d_{ij}(\mathbf{V})$ for each bank j, which represents the total amount of money that bank j pays out.

4.3 Implementation Details

We turn our attention to the implementation details of the algorithm. We have implemented only one algorithm for financial systems with both equity and debt contracts, which uses the same environment as the algorithms presented in Section 3.3.

However, the implementation of this algorithm is more complex than the others due to the close interconnections between the terms. Therefore, it is essential to carefully follow the flow of the algorithm as presented in the pseudocode of Algorithm 3. Firstly, we draw inspiration from the Fictitious Default Algorithm and initialize the $Default\_set$ to track the indices of defaulting banks and monitor the Boolean variable $isDefault$ to check if any banks are still defaulting, which serves as a termination condition for the algorithm. Then, we initialize the starting point as the upper bound for $\mathbf{V}$, denoted as $\mathbf{V}^0$, which combines all of the nodes' primitive assets, all the money they should receive, and their equity values, all of which are constant and can be easily calculated.

Next, we enter the while loop with the upper bound $\mathbf{V}^0$ and continue iterating to update $\mathbf{V}$ according to Equation 4.7 until no more banks are defaulting. Although we update $\mathbf{V}$ based on the value of $\mathbf{V}$ from the last iteration, the terms are all closely interrelated and dependent on the value of $\mathbf{V}$ and the $Default\_set$. These terms include $\mathbf{S}(\mathbf{V})$, $\mathbf{d}^A(\mathbf{V})$, and $\mathbf{b}(\mathbf{V},\mathbf{p})$, while $\mathbf{I}$ is the identity matrix and the product of $\mathbf{qp}$ and the $\mathbf{D}^L$ are all constant throughout the process.

For the term $\mathbf{S}(\mathbf{V})$, the matrix $\mathbf{S}$ changes depending on the banks that are in default under $\mathbf{V}$. We will discuss the updating process for matrix $\mathbf{S}$ later. Once we have the updated matrix $\mathbf{S}$, we calculate the inverse of the square matrix $\mathbf{I}-\mathbf{S}$ using the `numpy.linalg.inv` function.

The computation of the terms $\mathbf{d}^A(\mathbf{V})$ and $\mathbf{b}(\mathbf{v},\mathbf{p})$ can be quite complicated, as both are dependent on $\mathbf{V}$, the $default\_set$, and each other. To handle these dependencies effectively within the algorithm, we define several helper functions.

The first helper function, which we name `j_pay_i`, calculates the money that bank j needs to pay to bank i (i.e., $d_{ij}(\mathbf{V})$), given the input $\mathbf{V}$ and the $default\_set$. The equations for calculating this value are presented in Equation 4.4 and Equation 4.5. Two situations are considered here, and we differentiate between them by checking if bank j is in the $default\_set$:

- If bank j is not in default, it should pay bank i the nominal liability, which is initialized in matrix $\mathbf{D}$.
- However, if bank j is in default, the payment value is calculated by multiplying a relative percentage by the total assets of bank j minus the bankruptcy costs it incurs. Upon closer observation, the complex expression of the first term inside the maximization in Equation 4.4 can be simplified in our implementation as follows:

$$\begin{aligned}&\sum_k q_{jk}p_k + d_j^A(\mathbf{V}) + \sum_h S_{jh}V_h^+ - \beta_j(\mathbf{V},\mathbf{p})\\ &\rightarrow \mathbf{V}_j + D_j^L\end{aligned}$$

The second helper function, which we name `bank_total_received`, calculates the total value that a bank i could receive under the given $\mathbf{V}$. This function takes the index of the bank $\mathbf{V}$ and $Default\_set$ as inputs and outputs the total value that bank i would receive from all other banks. To achieve this, we make use of the previously defined `j_pay_i` function to determine the value that bank i could obtain from each other bank, and then return the sum of all these received values.

The third helper function, which we name `compute_total_received_vector`, takes the inputs $\mathbf{V}$ and $Default\_set$ and outputs the vector representing $\mathbf{d}^A(\mathbf{V})$, where each value in the vector is computed using the `bank_total_received` function for each bank. By utilizing the first three helper functions, we can now effectively compute the term $\mathbf{d}^A(\mathbf{V})$ in Equation 4.7.

The fourth helper function, named `bank_equity_received`, calculates the total equity value that a node could receive depending on the inputs $\mathbf{V}$ and $Default\_set$. For bank i, it iterates through all other banks to add up their respective values. For each bank j, two separate situations need to be considered:

- If bank j is not in $Default\_set$, the value that bank i could receive is equal to S_{ij} * max(V_j, 0).
- Otherwise, if bank j is in default, bank i receives nothing from bank j. The function then returns the sum of all received equity values for bank i.

The fifth helper function, named `bank_bankruptcy_cost`, takes the same inputs as in the function `bank_total_received`, i.e., the index i, $\mathbf{V}$, and $Default\_set$. The function outputs the bankruptcy cost incurred by bank i under $\mathbf{V}$ using the equations for the bankruptcy cost presented in Equation 4.6. Therefore, we also implement this function differently in two situations:

- If bank i is not in the $Default\_set$, we simply return 0 as there is no bankruptcy cost.
- Otherwise, if bank i is in default, we calculate the bankruptcy cost it incurs according to Equation 4.2. The term b is a fixed cost that was initialized. The second part is a percentage a multiplied by the sum of the assets, where the sum of the assets is calculated by adding together the primitive asset that bank i owns, the equity value of bank i that could be received under the current $\mathbf{V}$ (obtained using the helper function `bank_equity_received`, and the total debt value that bank i could receive (calculated using the helper function `bank_total_received`).

Finally, the sixth helper function is named `compute_bankruptcy_costs_vector`, calculates the bankruptcy cost for all banks given the current $\mathbf{V}$ and the $Default\_set$. To do this, we iterate through all indices of the banks, and pass the inputs to the helper function `bank_bankruptcy_cost` to calculate the bankruptcy cost for each bank individually. We then assemble these values into a vector. As a result, the function outputs a vector of bankruptcy costs, $\mathbf{b}(\mathbf{V}, \mathbf{p})$.

With these helper functions, we can now easily update the terms $\mathbf{d}^A(\mathbf{V})$ and $\mathbf{b}(\mathbf{V}, \mathbf{p})$ using the functions `compute_total_received_vector` and

`compute_bankruptcy_costs_vector`, respectively. To do this, we simply provide the functions with $\mathbf{V}$ and $Default\_set$ as inputs.

After updating $\mathbf{V}$ for this round, we now proceed to update the $Default\_set$ by adding the indices of the defaulting nodes into the set. The criterion here should be the comparison in Equation 4.6, which compares the total asset of the bank under $\mathbf{V}$ and its total nominal liability, and it is said to be in default as long as the total asset is smaller than its liability. In implementation, since we are comparing $\sum_k q_{ik}p_k + \sum_j S_{ij}V_j^+ + d_i^A(\mathbf{V})$ and D_i^L, we can also simplify it as:

$$\begin{aligned}&\sum_k q_{ik}p_k + \sum_j S_{ij}V_j^+ + d_i^A(\mathbf{V}) - D_i^L \\ &\rightarrow \mathbf{V}_j - \mathbf{b}_i(\mathbf{V},\mathbf{p}).\end{aligned}$$

If the result of $\mathbf{V}_i - \mathbf{b}_i(\mathbf{V},\mathbf{p})$ is positive, this indicates that the assets are sufficient to cover its total liabilities. Therefore, the bank is not in default and should not be added to the $Default\_set$. On the other hand, if the result is negative, the bank is in default and should be added to the set. It is important to note that not all banks can be in default at the same time. Therefore, we need to monitor the number of defaulting banks. If only one bank is left that is not in default, the algorithm should terminate.

Lastly, once the equilibrium value is found, determining the clearing payment vector becomes straightforward. Each value in the vector now simply equals to $\sum_i d_{ij}$. This can be effortlessly computed by iteratively applying the function `j_pay_i` for each bank.

4.4 Experiment

As we have only implemented the algorithm in one way, we are unable to compare its performance with alternative implementations as we did in the previous financial model. Nevertheless, we can conduct an initial experiment by applying the algorithm to a network with only three banks and see the results. Subsequently, we plan to investigate the algorithm's computational efficiency by measuring the time it takes to compute for different numbers of banks in the financial network.

4.4.1 A simple example

Motivation: Here, we would conduct a simple example with 3 banks in the financial network so that we could compute the result by ourselves, and compare with the result computed by the implemented algorithm.

Experiment settings: The necessary data for initialization includes the liability matrix **D**, the equity matrix **S**, the fixed cost vector **b**, the discount fraction vector **a** for the bankruptcy cost, and the value of primitive asset vector which equals to **qp**. We could initialize this product directly since it is constant throughout the algorithm. Here, we initialize a financial network of three banks, and the matrix **D** is:

$$\mathbf{D} = \begin{bmatrix} 0 & 376 & 38 \\ 0 & 0 & 0 \\ 0 & 186 & 0 \end{bmatrix}$$

the matrix **S** is:

$$\mathbf{S} = \begin{bmatrix} 0 & 0.99 & 0.56 \\ 0 & 0 & 0.44 \\ 0 & 0.01 & 0 \end{bmatrix}$$

the vector **b** is [0, 7, 9], the vector **a** is [0, 0,82, 0,07] and the primitive asset vector is [0, 602, 567].

Results: The initialized data is input into the Algorithm 3 in Section 4.2.2, and it outputs **V** = [1169, 356.1671354, 718.56167135] after one round since no banks defaulted. To verify if this is correct, we calculate **V** step-by-step. Here, using Equation 4.1, the bank value of Bank 0, Bank 1, and Bank 2 should be:

- $V[0] = p[0] + S[0][1]V[1] + S[0][2]V[2] + D[0][1] + D[0][2]$
- $V[1] = p[1] + S[1][2]V[2] - D[0][1] - D[2][1]$
- $V[2] = p[2] + S[2][1]V[1] - D[0][2] + D[2][1]$

Computing these elements by hand also gives **V** = [1169, 356.1671354, 718.56167135].

4.4.2 Elapsed time for different number of banks

Motivation: To see the time efficiency of the algorithm, we plan to conduct experiments on varying the number of banks in the financial network and analyzing the corresponding computational performance.

Experiment settings: For the data initialization, the same functions and methods are used as in Section 3.4.1. In more details, random seed of 45 is used here, the matrices we initialized include the liability matrix **D** (it also follows the core-periphery structure here), the equity matrix **S**, the fixed cost vector **b** and the discount fraction vector **a** for the bankruptcy cost, and the value of primitive asset vector. The number of banks n is set to 50, 100, 150, 200, 250, 300, 350 and 400 respectively.

Results: From the result, we could see a significant increase in elapsed time as the number of banks in the network grows larger, with a sharp increase observed after 250 banks, the elapsed time for 300 banks exceeds 1000 seconds, and the curve of the relationship between elapsed time and number of banks appears to follow a potentially exponential trend. The results of our experiments are summarized in Table 4.1 below. The elapsed time is reported in seconds and rounded to four decimal places for clarity. The visualization for the results is presented in Figure 4.2.

No. of Banks (n)	Elapsed time(s)
50	0.2862
100	1.8272
150	5.9764
200	13.52977
250	27.3672
300	1005.6980
350	1341.5660
400	2959.2157

Table 4.1: Comparison of the elapsed time for varying numbers of banks.

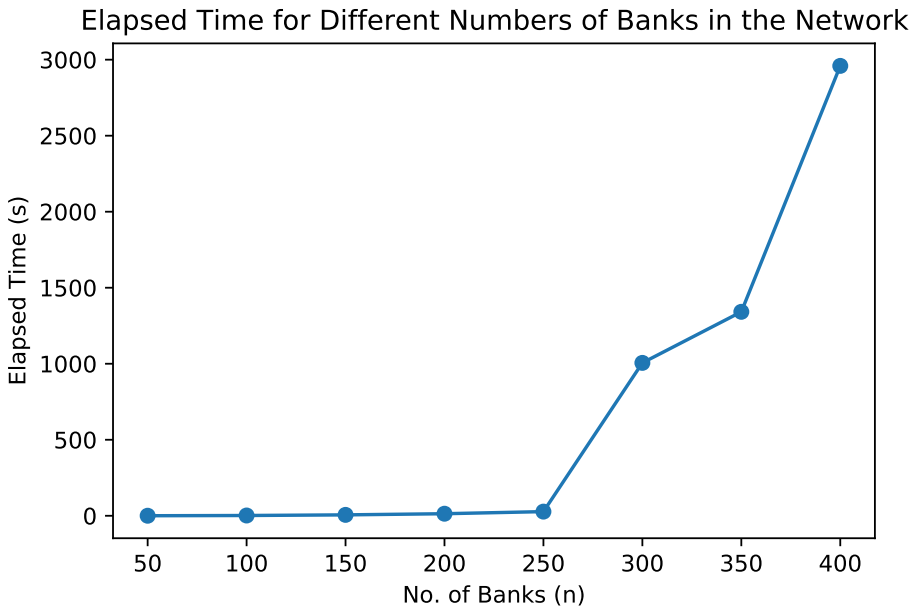

Figure 4.2: Comparison of Elapsed Time in seconds with varying number of banks (n).

Chapter 5

Conclusions

5.1 Conclusion

In this project, we summarized the financial model developed by Eisenberg and Noe and provided proof of their algorithm's efficacy [9]. We then implemented three algorithms in Python for finding the clearing payment vector in a financial network with debt contracts only, which, to the best of our knowledge, has not been done before. Moreover, we developed a method to generate reasonable data in the absence of an available data set, and conducted experiments to compare the performance of the algorithms in terms of speed and accuracy under different conditions. Our results showed that the algorithm based on the concept of linear programming provided the optimum solution, but its computational time increased significantly as the number of nodes increased. On the other hand, the algorithm based on pure Value Iteration was the fastest, while the Fictitious Default Algorithm provided valuable information for evaluating the risk. Then, we extended the model to include equity contracts, which was based on the model developed by Jackson *et al.*. Finally, we implemented the algorithm to determine the equilibrium banks' values in Python for the first time.

5.2 Limitations

First and foremost, the limited availability of real-world financial network datasets greatly restricts the exploration of both algorithms. Although we have learned about the characteristics of financial network structures from other literature and attempted to simulate real financial networks, the structure in each region or country varies significantly due to differences in financial systems and policies. These differences may also impact our analysis of the algorithms.

Next, the algorithm for finding the equilibrium banks' value, which we used, is based on relatively new literature published in 2020 [14], with limited prior research on it. Furthermore, since the terms are all interrelated and the interconnectedness is extremely close, updating each term during each round of the algorithm proved more challenging than anticipated. As a result, only one implementation was carried out, thereby limiting

our ability to explore its potential and conduct further related experiments.

5.3 Future Work

Therefore, a potential future direction of this work is to implement alternative versions of the algorithm for determining the equilibrium banks' value and conduct experiments to compare the performance and results between the algorithms. In addition, as demonstrated in our experiments in Section 4.4.2, the elapsed time of the algorithm greatly increases when the number of banks increases, therefore another direction is to advance the algorithm to increase the computational efficiency.

Eisenberg and Noe's financial model has been extended to various cases, such as the continuous-time model presented by Banerjee *et al.* [2], and under the assumption that firms may not make partial payments, as discussed by Bardoscia *et al.* [3]. As a future direction, algorithms for these extensions could be implemented and tested using the current implementation as a foundation.

Bibliography

[1] Research data — deutsche bundesbank.

[2] Tathagata Banerjee, Alex Bernstein, and Zachary Feinstein. Dynamic clearing and contagion in financial networks. *arXiv preprint arXiv:1801.02091*, 2018.

[3] Marco Bardoscia, Gerardo Ferrara, Nicholas Vause, and Michael Yoganayagam. Full payment algorithm. *Available at SSRN 3344580*, 2019.

[4] Garrett Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940.

[5] Uwe Blien and Friedrich Graef. Entropy optimizing methods for the estimation of tables. In *Classification, Data Analysis, and Data Highways: Proceedings of the 21st Annual Conference of the Gesellschaft für Klassifikation eV, University of Potsdam, March 12–14, 1997*, pages 3–15. Springer, 1998.

[6] Joao F Cocco, Francisco J Gomes, and Nuno C Martins. Lending relationships in the interbank market. *Journal of Financial Intermediation*, 18(1):24–48, 2009.

[7] Ben Craig and Goetz Von Peter. Interbank tiering and money center banks. *Journal of Financial Intermediation*, 23(3):322–347, 2014.

[8] Hans Degryse and Gregory Nguyen. Interbank exposure: An empirical examination of systemic risk in the belgian banking system. 2004.

[9] Larry Eisenberg and Thomas H. Noe. Systemic Risk in Financial Systems. *Management Science*, 47(2):236–249, February 2001. Publisher: INFORMS.

[10] Matthew Elliott, Benjamin Golub, and Matthew O Jackson. Financial networks and contagion. *American Economic Review*, 104(10):3115–3153, 2014.

[11] Charles Albert Eric Goodhart and CAE Goodhart. *Institutional Separation between Supervisory and Monetary Agencies (1993) (with Dirk Schoenmaker)*. Springer, 1995.

[12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[13] Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[14] Matthew O. Jackson and Agathe Pernoud. Investment Incentives and Regulation in Financial Networks, March 2019.

[15] Matthew O Jackson and Agathe Pernoud. Credit freezes, equilibrium multiplicity, and optimal bailouts in financial networks. *arXiv preprint arXiv:2012.12861*, 2020.

[16] Matthew O. Jackson and Agathe Pernoud. Systemic Risk in Financial Networks: A Survey. *Annual Review of Economics*, 13(1):171–202, 2021. _eprint: https://doi.org/10.1146/annurev-economics-083120-111540.

[17] Jiří Matoušek and Bernd Gärtner. *Understanding and using linear programming*, volume 33. Springer, 2007.

[18] Steffen Schuldenzucker, Sven Seuken, and Stefano Battiston. Finding clearing payments in financial networks with credit default swaps is ppad-complete. *LIPIcs: Leibniz International Proceedings in Informatics*, (67), 2017.

[19] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

[20] Christian Upper. Simulation methods to assess the danger of contagion in interbank markets. *Journal of financial stability*, 7(3):111–125, 2011.

[21] Christian Upper and Andreas Worms. Estimating bilateral exposures in the german interbank market: Is there a danger of contagion? *European economic review*, 48(4):827–849, 2004.

[22] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[23] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.