Algorithms for Computing Clearing Payments in Financial Networks

Minsuan Teh

Master of Science
School of Informatics
University of Edinburgh
2024

Abstract

This research examines the performance of various algorithms designed to determine optimal clearing payment vectors in financial systems. The models by Eisenberg and Noe, along with M. O. Jackson and A. Pernoud, are implemented and evaluated. The study highlights the effectiveness of the basic iteration algorithm and outlines the factors influencing the bankruptcy rate in financial systems. It also explores the specific performance patterns of Dang, Qi and Ye's algorithm, especially in scenarios with solvent banks. The unique behaviour of Dang, Qi and Ye's algorithm, particularly its dependency on the convergence rate of the monotone function, is discussed. A substantial portion of the paper provides Python-based implementations, connecting theory with practical application. The findings offer a nuanced understanding of financial systems and have potential implications for both academia and the financial industry.

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.

(*Minsuan Teh*)

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Professor Kousha Etessami, for his unwavering support and guidance throughout this project. His expertise and timely answers were instrumental in the successful completion of my research.

I also wish to acknowledge Overleaf (https://www.overleaf.com/), a cloud-based LaTeX editor, which was indispensable in the preparation of my paper. The entire writing process was facilitated through this platform.

Furthermore, I would like to recognize Grammarly (https://app.grammarly.com/), a cloud-based typing assistant. Grammarly's web browser plugin automatically detected and corrected typing errors, significantly enhancing the accuracy and clarity of my dissertation as I composed it within Overleaf.

Table of Contents

1 Introduction **1**

1.1 Related Work 1

1.2 Motivation 3

1.3 Structure and Summary 4

2 Framework and Algorithms **6**

2.1 Economic Framework by Eisenberg and Noe 6

2.1.1 Basic Iteration Algorithm (BIA) 7

2.1.2 Fictitious Default Algorithm (FDA) 7

2.1.3 Linear Programming (LP) 8

2.1.4 Uniqueness of Clearing Payment Vectors 9

2.2 Economic Framework by Jackson and Pernoud 10

2.2.1 The Algorithm by Jackson and Pernoud 11

3 The Implementation **12**

3.1 The Model by Eisenberg and Noe 13

3.1.1 Implementation of Basic Iteration Algorithm 14

3.1.2 Implementation of Fictitious Default Algorithm 16

3.1.3 Implementation of the Linear Programming ALgorithm 17

3.1.4 Implementation of the Algorithm by Dang, Qi and Ye 17

3.2 The Model by Jackson and Pernoud 18

3.2.1 Bankruptcy Cost 20

3.2.2 Implementation of the Algorithm by Jackson and Pernoud . . 21

3.3 Proof of Correctness 22

4 Experimental Evaluation of the Implementation **24**

4.1 Bankruptcy Rate 25

4.1.1 `liability` 25

4.1.2 `asset` . . . 26
4.1.3 `edge` . . . 27
4.2 Complexity of the Financial System . . . 28
4.2.1 Effect of Varying Complexity on LP . . . 28
4.2.2 When Banks are Mostly Solvent . . . 29
4.2.3 When Banks are Mostly Defaulting . . . 30
4.3 The Algorithm by Jackson and Pernoud . . . 30
4.3.1 Comparing with the Other Algorithms . . . 31
4.3.2 Multiple Assets and Equity Shares . . . 32
4.3.3 Bankruptcy Cost . . . 34
4.4 The Algorithm by Dang, Qi and Ye . . . 36

5 Conclusion **38**
5.1 Future Work . . . 39

Bibliography **40**

A Python Implementaion Code **42**
A.1 financialSystem.py . . . 42
A.2 fictitiousDefaultAlgorithm.py . . . 45
A.3 LPSolver.py . . . 46
A.4 DQY_Algorithm.py . . . 47
A.5 financialSystemWithShares.py . . . 49

Chapter 1

Introduction

The global financial system has become increasingly interconnected, resulting in complex financial obligations between institutions. This interconnectedness has significant implications for the stability of the financial system, as distress in one institution can propagate through the network, potentially leading to systemic risk and financial crises. One essential aspect of understanding and managing these risks is the computation of clearing payments in financial networks. Clearing payments represent the amounts that financial institutions can pay to their counterparties, considering the potential for default and the interconnected nature of financial obligations.

The general scope of this project encompasses the development, analysis, and comparison of various algorithms for computing clearing payments in financial networks. We will focus on the theoretical foundations and the practical applications of these algorithms, assessing their performance, accuracy, and efficiency in different network scenarios.

1.1 Related Work

In 2001, Eisenberg and Noe published a paper that provides a comprehensive analysis of systemic risk in financial systems [1], focusing on the interconnectedness of financial entities and the impact of cyclical obligations. The paper presents a model that includes existence-uniqueness results and characterizations of clearing vectors, which are used to understand the distribution of value in a multi-firm environment with cyclical obligations. The authors argue that a unique division of value, consistent with standard rules of value division, always exists under certain regularity conditions. The authors also discuss the economic framework of their model, considering an economy populated by distinct

financial nodes, each with nominal liabilities to other entities in the system. They represent this structure of liabilities with a nominal liabilities matrix and an operating cash flow vector, which together form a financial system. The paper further explores the implications of limited liability and absolute priority, noting that these principles imply that every node pays the minimum of what it has and what it owes. The authors establish that every financial system has a clearing vector, which is a fixed point of a map defined in the paper. The authors also consider potential extensions to their model, such as allowing for violations of absolute priority, incorporating true dynamics by allowing for more than one clearing date, and introducing uncertainty into the framework.

A paper by M. O. Jackson and A. Pernoud explores the impact of connections between financial institutions on their investment incentives and discusses optimal regulation in a network setting [2]. The paper also highlights the role of both debt and equity contracts in financial networks, emphasizing their prevalence in practice and their distinct influences on investment incentives. The paper references several other works, including studies by Shu [3], Zawadowski [4], Galeotti and Ghiglino [5], and Vohra, Xing, and Zhu [6], which explore various aspects of network externalities, investment risks, and agency conflicts within financial institutions. The authors of this paper argue that their model of financial networks, incorporating both debt and equity contracts, generalizes existing models used to understand systemic risk. The paper also discusses the implications of limited liability for banks, noting that it can lead to increased risk-taking and investment correlation. The authors argue that due to financial interdependencies, bank values depend positively on each other, inducing complementarities in their returns to investments. The authors also delve into the technical aspects of their model, discussing the conditions under which a bank defaults and the associated bankruptcy costs. They highlight that the value of a bank is weakly increasing in that of others in the network, ensuring the existence of a solution to their model's equations.

In the paper by S. Schuldenzucker, S. Seuken and S. Battiston, the authors present a comprehensive study on the complexity of finding clearing payments in financial networks that incorporate credit default swaps (CDS) [7]. The authors delve into the intricacies of the financial system, particularly focusing on the role of CDS in the network. Credit Default Swaps are financial derivative instruments that provide insurance against the risk of a default by a borrower. They play a crucial role in financial networks, acting as a form of insurance for lenders. The authors highlight the importance of understanding the complexity of finding clearing payments in such

networks, as it is critical to financial stability. The paper's main contribution is proof that the problem of finding clearing payments in financial networks with CDS is PPAD-complete. This means that the problem is as hard as the hardest problems in the class PPAD, which is a class of problems known for their computational complexity. The authors provide detailed proof to support this claim, contributing to the understanding of the computational complexity of problems in financial networks. The authors also discuss the implications of their findings. The PPAD-completeness result implies that unless P=PPAD (which is considered unlikely by most computer scientists), there is no polynomial-time algorithm that can find clearing payments in all cases. This has significant implications for the efficiency and stability of financial systems.

S. D. Ioannidis, B. de Keijzer, and C. Ventre take an in-depth look into the intricacies of financial networks, notably those incorporating financial derivatives. [8]. The authors explore the concept of strong approximations and irrationality within these networks, shedding light on the intricate dynamics that govern financial systems. The paper also presents a series of mathematical gadgets, including division and multiplication gadgets. These gadgets are used to represent and analyze complex financial interactions within the network. However, the authors note that these gadgets do not satisfy non-degeneracy, a property that ensures the financial system's stability. This observation is significant as it highlights the inherent complexities and potential instabilities in financial networks involving financial derivatives. The authors also delve into the concept of fragment cycles, which are graphs representing the cyclical nature of financial interactions within the network. Understanding these cycles is crucial for predicting and managing financial contagion, a scenario where financial distress spreads across the network.

1.2 Motivation

The growing complexity and interconnectedness of financial institutions have led to an urgent need for efficient and accurate algorithms for computing clearing payments in financial networks. Clearing payments are crucial for mitigating systemic risk and ensuring financial stability, as they determine the amounts that institutions can pay to their counterparties while accounting for potential defaults and the interconnected nature of financial obligations. The gap that needs to be filled in the literature is a comprehensive analysis and comparison of existing algorithms for computing clearing payments and evaluating their performance, accuracy, and efficiency across various financial network scenarios.

Despite the availability of several algorithms in the literature, there is a lack of research that systematically investigates and compares their effectiveness in addressing the clearing payment problem in diverse contexts. Thus, this study is performed to identify the most suitable algorithms for computing clearing payments across various types of financial networks. In doing so, we consider the strengths and weaknesses of these algorithms and provide recommendations for practitioners, regulators, and policymakers.

Throughout this paper, the following research question will guide our study: ”What are the most effective algorithms for computing clearing payments in financial networks, and how do their performance, accuracy, and efficiency vary across different financial network scenarios?” By answering this question, our research aims to fill the existing gap in the literature and provide valuable insights into the most appropriate algorithms for computing clearing payments, ultimately contributing to the mitigation of systemic risk in interconnected financial systems.

1.3 Structure and Summary

This study has investigated multiple algorithms within financial systems, focusing on their theoretical foundations, practical implementations and performance evaluations. This section synthesises the major findings and contributions:

1. Introduction and Framework (Chapters 1-2):
   The study commenced with an introduction to the research problem and a detailed investigation of the mathematical framework. Algorithms by authors such as Jackson and Pernoud were analyzed, laying the groundwork for understanding financial contracts, bank values, default conditions, and payment dynamics.

2. Implementation (Chapter 3):
   The chapter provided a thorough implementation of financial systems, introducing classes and functions for validation and optimization of clearing payment vectors. This chapter also highlighted the importance of both accuracy and optimality in computational solutions.

3. Experimental Evaluation (Chapter 4):
   A comprehensive evaluation of various algorithms was conducted, with key findings as follows. BIA(top) Outperforms other algorithms most of the time,

making it a preferred choice. BIA(bottom) offers faster performance compared to other algorithms but falls short of BIA(top) in overall effectiveness. LP can be faster than BIA(top) when the bankruptcy rate is high. FDA may outpace LP but BIA(top) still prevails in these scenarios. JP is slower than FDA, BIA and LP but has the unique ability to compute the best clearing payment matrix, detailing the clearing payments between banks. DQY is the slowest in nearly all scenarios, but its theoretical complexity suggests potential advantages. The inclusion of equity shares, bankruptcy cost and multiple assets in financial systems affects the bankruptcy rate and JP's performance.

4. Conclusion (Chapter 5):
   The conclusion details the study of algorithms within financial models by Eisenberg and Noe, and Jackson and Pernoud. Experiments reveal how various factors, such as bankruptcy rate and system complexity, influence algorithm performance. While the basic iteration algorithm often excels, the Jackson and Pernoud algorithm uniquely computes clearing payments between banks. The study also examines the effects of equity shares and bankruptcy costs and suggests future research directions, including algorithm refinement and exploration of new financial contexts.

Chapter 2

Framework and Algorithms

2.1 Economic Framework by Eisenberg and Noe

Consider a financial system which comprises $N = \{1, ..., n\}$ banks. Each node within this system may possess a certain quantity of primitive assets and nominal liabilities. Let e be a vector with N entries, where the ith entry, e_i, signifies the total primitive assets of node i. The nominal liability of a node can be represented by a matrix, denoted as L, where the (i, j)-th entry of the matrix indicates the value of i's liability towards j. Consequently, node i's total nominal liabilities, $\bar{p}_i$, equals $\sum_{j=1}^{n} L_{ij}$. It's important to note that no node has a nominal liability to itself, hence $L_{ii} = 0$ for all $i \in N$. We also assume that all the nominal liabilities and the primitive assets are nonnegative, $L_{ij} \geq 0$ and $e_i \geq 0$ for all $i, j \in N$. Therefore, a financial system can be defined as a pair of (L, e), consisting of the nominal liability matrix and the primitive assets of the banks.

Let Π represent a relative liabilities matrix of the financial system, where the (i, j)-th entry of the matrix, Π_{ij}, equals $\frac{L_{ij}}{\bar{p}_i}$ if $\bar{p}_i > 0$ and 0 otherwise. The financial system can now be represented as $(\Pi, \bar{p}, e)$. utilising the relative liabilities matrix, we can calculate that the total payments required to be made by all other banks to bank ii are equal to $\sum_{j=1}^{n} \Pi_{ij}^{T} p_j$, where p_j is the payment that is required to be made by bank j. A bank is defined as bankrupt if the value of its equity is negative, that is, $\sum_{j=1}^{n} \Pi_{ij}^{T} p_j + e_i - p_i < 0$. The problem of finding the best clearing payment can now be defined as identifying a vector $p^* \in [0, \bar{p}]$ in a financial system $(\Pi, \bar{p}, e)$ such that the following conditions are satisfied.

- Limited Liability: $p_i^* \leq \sum_{j=1}^{n} \Pi_{ij}^{T} p_j^* + e_i$
- Absolute Priority: $p_i^* = \sum_{j=1}^{n} \Pi_{ij}^{T} p_j^* + e_i$ or $p_i^* = \bar{p}_i$

2.1.1 Basic Iteration Algorithm (BIA)

In the economic framework described in Section 2.1, it is evident that p^* is a clearing payment vector of a financial system $(\Pi, \bar{p}, e)$ if and only if

$$\forall i \in N, p_i^* = min(\sum_{j=1}^{n} \Pi_{ij}^T p_j^* + e_i, \bar{p}_i) \tag{2.1}$$

According to the Knaster–Tarski theorem, given that f is a monotone function, there must exist at least a fixed point of f. Consequently, the best clearing payment vector can be identified by locating the greatest fixed point of equation 2.1. Let f be the monotone function in equation 2.1, and then we can define a sequence of payment vectors as follows.

$$p^j = f(p^{j-1});\ p^0 = \bar{p}. \tag{2.2}$$

As $\bar{p}$ is the biggest possible payment vector that need to be made by each bank, the first fixed point that is found using this method will invariably be the biggest fixed point and hence the best clearing payment.

In certain instances where the fixed point is proven to be unique (further details in Section 2.1.4), we can also use $p^0 = \bar{0}$ where $\bar{0}$ is a vector of 0. The performance of the algorithm will be influenced by this choice, as demonstrated in Chapter 4. Regardless of the chosen p^0, the algorithm will identify the best clearing payment vector in at most $(max(\bar{p}) - 1)n$ iterations where $max(\bar{p})$ is the biggest payment that is required to be paid by any bank.

2.1.2 Fictitious Default Algorithm (FDA)

Eisenberg and Noe proposed an alternative algorithm for determining the best clearing payment vector within a financial system. This algorithm begins with the assumption that each bank will initially pay its debt in full. If all banks can meet their debt obligations without entering bankruptcy, then the best clearing payment vector has been found. However, if not all banks can fully meet their debt obligations, the algorithm proceeds to calculate the maximum debt payment that the bankrupt banks can afford while keeping the clearing payments of the non-bankrupt banks constant. If all banks can meet their debt obligations in the second iteration, the algorithm concludes. If not, the algorithm repeats the previous process in the subsequent iteration until all banks can meet their debt obligations under the new clearing payment vector.

The Fictitious Default Algorithm bears a strong resemblance to the basic iteration algorithm. Both algorithms hinge on the iterative querying of a monotone function until

a fixed point is identified. For the Fictitious Default Algorithm, we first need to define a *n x n* diagonal matrix, $\Lambda(p)$, such that

$$\forall i, j \in N, \Lambda(p)_{ij} = \begin{cases} 1 & i = j \text{ and bank } i \text{ defaults under } p \\ 0 & \text{otherwise} \end{cases}$$

Next, let $FF_{p'}(p)$ be the monotone function employed in this algorithm.

$$FF_{p'}(p) = \Lambda(p')(\Pi^T(\Lambda(p')p + (I - \Lambda(p')\bar{p})) + e) + (I - \Lambda(p'))(\bar{p}) \tag{2.3}$$

It is important to note that p' differs from p in that p' is used to determine the banks that default under p' for the calculation of $\Lambda(p')$. The banks which are not defaulting under p' would have to make the full payment, as shown in the second term of the addition in equation 2.3. Conversely, the banks which default under p' are required to disburse their full value, assuming that the defaulting banks pay p and non-defaulting banks pay in full.

The fixed point can be identified by defining a sequence of payment vectors as per equation 2.2. This sequence of vectors is referred to as the fictitious default sequence by the authors [1]. The best clearing vector can be assuredly found following this sequence because $\Lambda(p)\Pi$ has a row sum of less than 1. This implies that equation 2.3 is a monotone function and, as per the Knaster-Tarski theorem, is guaranteed to have at least one fixed point. Unlike the basic iteration algorithm, which requires at most $(max(\bar{p}) - 1)n$ iteration to find the best clearing payment vector, this algorithm requires at most $n - 1$ iterations. This is because each iteration of the algorithm identifies the best clearing payment for at least one bank.

2.1.3 Linear Programming (LP)

In the paper by Eisenberg and Noe, they showed that a financial system can be modelled as a linear programming problem [1]. Finding the best clearing payment vector is equivalent to maximizing a weighted vector subject to the limited liability condition. Formally, the best clearing payment vector p^* of a financial system $(\Pi, \bar{p}, e)$ and a function $f : [\bar{0}, \bar{p}] \to R$ can be found by:

$$\begin{aligned} P(\Pi, \bar{p}, e, f): \quad & Maximize\ f(p) \\ subject\ to: \quad & p \leq \Pi^T p + e \end{aligned}$$

In order to show that any solution to $P(\Pi, \bar{p}, e, f)$ is a clearing payment vector for the financial system, we have to make sure that the solution given by $P(\Pi, \bar{p}, e, f)$

satisfies limited liability and absolute priority mentioned in Section 2.1. Assuming that p^* is a solution to $P(\Pi, \bar{p}, e, f)$, then limited liability is satisfied by the feasibility of p^* in the programming problem. If the absolute priority condition were not satisfied, for instance at node i, then it must be the case that $p_i^* < \bar{p}_i$ and $(\Pi^T p^* + e - p^*)_i > 0$. Consider a vector p^ε which is equal to p^* in all dimensions except i, and for i, is given by $p_i^* + \varepsilon$, where ε is chosen to be sufficiently small to ensure that limited liability remains satisfied. p^ε is a feasible solution to the programming problem because:

$$(\Pi^T p^\varepsilon + e - p^\varepsilon)_j - (\Pi^T p^* + e - p^*)_j = \varepsilon \Pi_{ij} \geq 0 \tag{2.4}$$

Given that f is a strictly increasing function and p^ε is greater than p^* in one of its dimensions and p^ε is at least equal to p^* in other dimensions, we can conclude that $f(p^*) < f(p^\varepsilon)$, contradicting the assumption that p^* is a solution to $P(\Pi, \bar{p}, e, f)$.

2.1.4 Uniqueness of Clearing Payment Vectors

Eisenberg and Noe have established that under certain conditions, a financial system can have a unique clearing payment vector. To understand these conditions, we first need to define a few key terms.

Surplus Set. A set S of N is referred to as a surplus set if no node within the set has any obligations to any node outside the set and the set has positive operating cash flows. Formally, $\forall (i, j) \in S \times S^c, \Pi_{ij} = 0$ and $\sum_{i \in S} e_i > 0$.

Financial Structure Graph. This is a directed graph representing a financial system $(\Pi, \bar{p}, e)$. The vertices of the graph are the banks in the financial system N, and the edges are defined by $i \to j \Leftrightarrow \Pi_{ij} > 0$.

Risk Orbit. The risk orbit of a bank i, denoted as $o(i)$, is the set of all banks $j \in N$ such that there exists a directed path from i to j.

A financial system is deemed regular if every risk orbit, denoted as $o(i)$, constitutes a surplus set. In such a regular financial system, the maximum and minimum clearing payment vectors are the same. A simple way to ensure regularity is that all banks possess positive operational cash flows. Another straightforward criterion to ensure regularity is the close interrelation of all nodes in the financial structure graph, with at least one bank boasting a positive equity value. Given that the uniqueness of clearing payment vectors in a financial system can be ensured, it becomes feasible to employ other algorithms designed to find any fixed point of a monotone function. While it may

not be entirely feasible for a realistic financial system to satisfy any of the aforementioned conditions, it is worthwhile to examine how an algorithm with a theoretically faster runtime performs in reality.

In 2020, Dang, Qi, and Ye introduced an algorithm capable of identifying Tarski's fixed point within $O(log^d N)$ iterations [9]. The algorithm operates by iteratively refining an initial approximation of the fixed point. In each iteration, the algorithm updates the approximation based on the current estimate and the system's properties. The algorithm continues this process until it reaches a point where the estimate no longer changes significantly, which is the fixed point. Concurrently, Fearnley, Palvolgyi, and Savani demonstrated another algorithm that can locate Tarski's fixed point in $O(log^{2\lceil k/3 \rceil} N)$ iterations [10]. The algorithm operates in a similar iterative manner, but the update rule is different, leading to a different complexity. This study implements and compares the algorithm proposed by Dang, Qi, and Ye against other algorithms, as detailed in Section 4. The specifics of the algorithms are not discussed here as they fall outside the scope of this study, but they can be found in the original paper by Dang, Qi, and Ye [9].

2.2 Economic Framework by Jackson and Pernoud

In Section 2.1, the Eisenberg and Noe model was introduced, which posits simple debt contracts between banks and assigns initial endowments (or operating cash flows) to each bank. This section will build on that model by introducing the model proposed by Jackson and Pernoud [2]. This model expands on the previous one by allowing each bank to own multiple assets and a certain number of shares in other banks.

A financial system made up of $N = \{1, ..., n\}$ banks can be represented by a pair (L, e), as discussed in Section 2.1. In reality, a bank can invest in a variety of primitive assets. As a result, a single endowment for each bank may not accurately reflect reality. Let $K = \{1, ..., k\}$ denote the collection of primitive assets, each with a market price of a_k. Let q be a matrix with the (i,k)-th entry representing the amount of asset k invested by bank i. Then, the total value of bank assets i can then be represented by $\sum_{j=1}^{j=k} q_i j a_j$.

A new matrix, S, is introduced to represent the number of shares that each bank owns in other banks. For example, $S_{ij} \in [0,1]$ represents the number of equity shares owned by bank i in bank j. An extra node is introduced into the financial system to prevent nonsensical cycles in which each bank is entirely owned by others in the cycle. This node, node 0, ensures that $\forall i, j \in N, S_{0i} = 1 - \sum_{j=1}^{j=n} S_{ji} > 0$. Note that no bank can own an equity share in node 0 because it represents outside investors, $\forall i \in N, S_{i0} = 0$.

Because of the addition of node 0, L is now a $(n+1) \times (n+1)$ matrix.

2.2.1 The Algorithm by Jackson and Pernoud

Jackson and Pernoud's work investigates the links formed by financial contracts between banks and introduces the concept of bank values, denoted as $V = \{V_1, ..., V_n\}$. They describe a bank as in default when the value of its assets is insufficient to meet its liabilities, resulting in bankruptcy costs that might vary depending on the health of other banks and the value of its investments, $\beta_i(V,a) \geq 0$. The authors also explore the notion of a debtor i making a payment to a creditor j that is equivalent to the face value of the debt if the debtor is solvent, $d_{ij}(V) = L_{ij}$. However, if the debtor fails, its creditors become the debtor's residual claimants on its assets, and their claims on the creditor are rationed correspondingly.

$$d_{ij}(V) = \Pi_{ij}\ max(\sum_k q_{ik}a_k + d_i^A(V) + \sum_h S_{ih}V_h^+ - \beta_i(V,a), 0) \tag{2.5}$$

where $V_h^+ \equiv max(V_h, 0)$ and $d_i^A(V) = \sum_j d_{ji}(V)$.

It's important to highlight that when bank j goes bankrupt, equity holders in bank j receive no payment, hence $S_{ij}V_j^+ = 0$. Due to the principle of limited liability, the value of bank i's equity stake in j cannot fall into the negative. Additionally, when a bank is solvent, it avoids incurring bankruptcy costs: $\beta_i(V,a) = 0$. This section will not delve into the specifics of the function to be used for bankruptcy costs; this will be discussed in Section 3.2.1. However, the chosen function for bankruptcy costs will satisfy the condition that $\sum_h d_{hi}(V) - \beta_i(V,a)$ is nondecreasing in V. This ensures that a bank's balance sheet does not deteriorate when the value of its counterparties increases. The bank values can be solved by the following equation:

$$V = (I - S(V))^{-1}([qa + d^A(V) - \bar{p}] - \beta(V,a)) \tag{2.6}$$

where $S_{ij}(V) = 0$ if j defaults according to V and $S_{ij}(V) = S_{ij}$ otherwise.

The algorithm begins by assuming that all banks are solvent and sets $d(V) = L$. It then goes through an iterative process, updating the payment vector using equation 2.5 and recalculating bank values using equation 2.6. When the bank values reach a state of equilibrium, that is, when a fixed point in V is found, the best clearing payment vector is established. The best payment vector may therefore be calculated as $p_i^* = \sum_j d_{ij}(V)$.

Chapter 3

The Implementation

Python was chosen as the implementation language due to several reasons.

Ease of Use. Python is known for its simplicity and readability, which makes it a great language for implementing complex algorithms and mathematical models.

Scientific Computing Libraries. Python has a rich ecosystem of scientific computing libraries such as NumPy and SciPy. The code uses NumPy for matrix operations and random number generation, and the linprog function from SciPy is imported. These libraries provide efficient and convenient functions for performing complex mathematical operations.

Performance. Despite being an interpreted language, Python can be quite fast when used with libraries like NumPy and SciPy, which are implemented in C and Fortran and can perform complex operations very efficiently [11].

Popularity in the Data Science Community. Python is a popular language in the data science and machine learning community [12]. This means that it's likely that any potential collaborators are familiar with the language, which makes development and maintenance easier.

The choice of Python packages is also justified by the needs of the project.

NumPy. Used for efficient numerical computations. It provides a high-performance multidimensional array object, and tools for working with these arrays. NumPy is used here for creating and manipulating arrays, which are used to represent vectors and matrices in the financial system.

Scipy. A library used for scientific and technical computing. It provides many efficient and user-friendly interfaces for tasks such as numerical integration, interpolation, optimization, linear algebra, and more. In this project, the linprog function from SciPy

is imported, which is a tool for linear programming.

timeit. A module for timing small bits of Python code. It has both command-line interface and a callable one. It avoids a number of common traps for measuring execution times.

matplotlib. A comprehensive library for creating static, animated, and interactive visualizations in Python. It provides a highly customizable and powerful interface for generating a wide variety of plots and charts. In the context of this project, matplotlib would be used for visualizing the results of the experiments and providing insights into the behavior of the implemented algorithms. For instance, it is used to plot the number of bankrupt entities as a function of the initial assets and liabilities, and plot the convergence of the clearing payment vector over iterations. These visualizations can help to understand the dynamics of the financial system and the effects of different parameters.

3.1 The Model by Eisenberg and Noe

The economic framework by Eisenberg and Noe which consists of a number of banks is implemented in **financialSystem.py**. Each bank has an operating cash flow and liabilities represented by vectors and matrices respectively.

```
1 class financialSystem():
2     def __init__(self, n, liability, asset, maxDistance=1, edge=0):
```

The implementation utilises a Python class named `FinancialSystem()` to emulate the financial system. This class requires various input parameters including the number of banks, the expected nominal liabilities and primitive assets for each bank, along with other parameters necessary for generating the financial system and calculating the clearing payment vector.

The `maxDistance` parameter serves as a convergence criterion within the iterative algorithms tasked with finding the best clearing payment vector. This parameter sets the maximum Euclidean distance allowed between successive iterations of the clearing payment vector, serving as a tolerance level for the algorithms to consider the vector as having converged to a solution. The `edge` parameter is used to specify the expected number of edges or connections each bank has to other banks in the financial system. All variables in `FinancialSystem()` class are shown in Table 3.1.

```
class financialSystem():
    def generate(self):
```

The `generate()` method in the `FinancialSystem()` class is used to construct a financial system based on the parameters set during initialization. Initially, it checks whether `n` is not smaller than 2, as a minimum of two banks is required to calculate the best clearing payment vector in the financial system. Following that, the method ensures that the `edge` variable falls within the range of 1 to `n - 1`. The `assetVector` variable is then generated using NumPy's `random.normal()` function with the mean to `asset` and the standard deviation as $\sqrt{n}$.

After that, the `liabilityMatrix` variable is generated as follows. A $n \times 1$ `seed` array is generated using `random.normal()` with `edge` as the mean and $\sqrt{n}$ as the standard deviation. Each entry of the array has a maximum value of `n-1` and a minimum value of 0. Now, for each row of `liabilityMatrix`, randomly choose some banks between 0 and `n-1` using to `random.choice()` with a size of `seed[i]` where `i` represents the current number of row of `liabilityMatrix`. Bank `i` will owe the chosen banks according to the numbers generated from `random.normal()` with mean as `liability` and standard deviation as $\sqrt{n}$. Lastly, `totalPaymentVector` and `relativeLiabilityMatrix` will be calculated using `liabilityMatrix`.

Note that $\sqrt{n}$ is used as the standard deviation of normal distribution because it provides a balance between the variability of the debts and the size of the network. This variability creates diverse scenarios in the network, allowing for more realistic modelling of financial systems. However, it's important to note that the use of $\sqrt{n}$ is a simplifying assumption. In real-world financial networks, the distribution of debt can be influenced by many factors and may not follow this standard deviation. It's also worth noting that the use of $\sqrt{n}$ can be adapted based on the specific characteristics or requirements of the model being used.

3.1.1 Implementation of Basic Iteration Algorithm

```
class financialSystem():
    def solve(self, topToBottom=True):
```

The algorithm is implemented within the `financialSystem()` class as a single function, `solve()`. The function accepts one argument, `topToBottom`, which initializes

Variables	Description
n	An integer that records the number of banks
clearingPaymentVector	A NumPy array that records the clearing payment vector
assetVector	A NumPy array that records the assets of each bank
liabilityMatrix	A 2-by2 NumPy array that records the nominal liability of banks
maxDistance	A floating point that sets the maximum Euclidean distance allowed between successive iterations of the clearing payment vector
edge	An integer that records the expected number of edges for each bank
asset	A floating point that records the expected amount of assets for each bank
liability	A floating point that records the expected amount of liabilities for each bank
relativeLiabilityMatrix	A 2-by-2 NumPy array that records the relative liability of each bank
totalPaymentVector	A NumPy array that records the total amount of debt that each bank needs to pay

Table 3.1: List of Variables in `financialSystem` class

the `clearingPaymentVector` to either an array of zeros or to the `totalPaymentVector`. Subsequently, the algorithm iterates until the best clearing payment vector is identified. In each iteration, a new payment vector is computed in accordance with equation 2.1. The `clearingPaymentVector` is then updated to be the element-wise minimum of the newly calculated payment vector and `totalPaymentVector`.

The algorithm compares the Euclidean distance between the new and old `clearingPaymentVector`. If this distance is less than `maxDistance`, the algorithm concludes and returns the new `clearingPaymentVector` as the best clearing payment vector. This comparison is performed using the `close()` function within the `financialSystem()` class, which checks if the Euclidean distance between two arrays is smaller than `maxDistance`.

To expedite calculations, the implementation leverage NumPy's `dot` function to compute the dot product between the transposed `relativeLiabilityMatrix` and the

old `clearingPaymentVector`. This approach is more efficient than computing the dot product element-wise with a `for` loop. The same principle of eschewing `for` loops when possible is applied throughout the implementation to maximize efficiency and is also extend to other NumPy functions like `transpose`, `sum`, `minimum` and more.

3.1.2 Implementation of Fictitious Default Algorithm

```
1 class fictitiousDefaultAlgorithm():
2     def __init__(self, financialSystem):
```

The algorithm is encapsulated within a separate class named `fictitiousDefaultAlgorithm()`. The class requires an instance of the `financialSystem()` class to determine the best clearing payment vector using the Fictitious Default Algorithm. It also incorporates two additional variables, as outlined in Table 3.2.

Variables	Description
defaultMatrix	A 2-by-2 NumPy array in which its diagonal element records the state of banks
financialSystem	An instance of the `financialSystem` class

Table 3.2: Two Additional Variables in the `fictitiousDefaultAlgorithm` class

```
1 class fictitiousDefaultAlgorithm():
2     def solve(self):
3     def updateDefaultMatrix(self):
4     def updatePaymentVector(self):
```

Much like the Basic Iteration Algorithm, the algorithm is implemented within a function called `solve()`. Initially, the function assigns the `totalPaymentVector` variable to the `clearingPaymentVector` variable. It then embarks on an iterative process to identify the best clearing payment vector. During each iteration, the `defaultMatrix` variable is updated using the `updateDefaultMatrix()` function. This update involves calculating the total assets or cash of each bank, then assigning either 1 or 0 to solvent or bankrupt banks, respectively, using the NumPy `where` function.

Following this, the `updatePaymentVector()` function is called to compute a new payment vector in accordance with equation 2.3. As previously noted, all calculations

involving the dot product utilise the NumPy `dot` function, avoiding the use of a `for` loop for improved performance. The algorithm ceases its operation if the `close()` function returns True. This occurs when the new payment vector and the old payment vector are sufficiently close to each other as determined by the pre-set tolerance.

3.1.3 Implementation of the Linear Programming ALgorithm

```
1 class LPSolver():
2     def __init__(self, financialSystem):
3     def solve():
```

As discussed in Section 2.1.3, the process of identifying the best clearing payment can be modeled as a maximization problem within linear programming. This approach is implemented in the `LPSolver()` class. The class requires an instance of `financialSystem` class as an argument during initialization.

The `LPSolver()` class features a single function named `solve()`. When this function is invoked, it determines and returns the best clearing payment vector. To solve the system of linear equations, the `optimize.linprog` function from the SciPy library is utilised as the linear programming solver. The solutions to this system are bounded by 0 and $\bar{p}$, ensuring that the resulting clearing payment vector adheres to the constraints of the financial system.

3.1.4 Implementation of the Algorithm by Dang, Qi and Ye

The algorithm put forth by Dang, Qi and Ye, which was previously implemented in a related study [13], serves as a significant influence for the current study's implementation. The algorithm is encompassed within a class named `DQY_algorithm()`. Like the other classes implemented in this project, `DQY_algorithm()` takes an instance of `financialSystem()` as an argument during initialization.

The class consists of five functions: `solve()`, `helper()`, `A_Below_B()`, `A_Above_B()`, and `updatePaymentVector()`. The `solve()` function, when invoked, identifies and returns the best clearing payment vector. The `helper()` function aids the `solve()` function in its operations. The `A_Below_B()` and `A_Above_B()` functions compare the positions of two arrays and return a boolean value based on the comparison. The `updatePaymentVector()` function updates the payment vector in line with equation 2.1.

```
class DQY_Algorithm():
    def __init__(self, financialSystem):
    def solve(self):
    def helper(self, depth):
    def A_Below_B(self, arrayA, arrayB):
    def A_Above_B(self, arrayA, arrayB):
    def updatePaymentVector(self):
```

In Dang, Qi and Ye's original paper, the algorithm operates on a complete lattice, thereby simplifying the comparison between two points. However, in the context of this study, the payment vector is represented by an array of type `float64`. While it is technically possible for two arrays of floating point numbers to be identical, this isn't a practical approach from a performance perspective. Therefore, two arrays are considered equivalent if the Euclidean distance between them across all dimensions is less than `maxDistance`.

Furthermore, a part of the algorithm involves the comparison of a single entry from each of the two arrays. In such cases, the entries are deemed equivalent if the absolute distance between them is less than `maxDistance/sqrt(n)`. This approach ensures that the Euclidean distance between the two full arrays remains less than `maxDistance`, even if all the individual entries have a Euclidean distance of `maxDistance/sqrt(n)`.

3.2 The Model by Jackson and Pernoud

The economic framework proposed by Jackson and Pernooud, which takes into account the ownership structure of banks, is implemented in the `financialSystemWithShares()` class. This framework extends the model by Eisenberg and Noe by incorporating the equity shares that banks have in one another, represented by a matrix.

```
class financialSystemWithShares(financialSystem):
    def __init__(self, n, k, liability, asset, costFunction, maxDistance=1,
    ↪  debtEdge=0, sharesEdge=0, quantityEdge=0, financialSystem=None):
```

Similar to the `financialSystem()` class, `financialSystemWithShares()` class also requires various parameters during initialization including the number of banks and assets, among others. Notably, there are additional parameters to account for the ownership structure, as outlined in Table 3.3, such as the number of different types of

assets `k`, and `sharesEdge` which indicates the expected number of shares each bank holds in other banks.

Variables	Description
k	An integer that represents the number of different types of assets in the financial system
quantity	A n-by-k NumPy array that records the number of assets that each bank holds
quantityEdge	An integer that records the expected number of assets that each bank holds
shares	A n-by-n NumPy array that records the number of shares that each bank holds on other banks
sharesEdge	An integer that records the expected number of shares that each bank holds on other banks
valueVector	A n-by-1 NumPy array that records the value of each bank
solventMatrix	A n-by-n NumPy array which is equivalent to identity matrix - `defaultMatrix`

Table 3.3: Additional Variables in the `financialSystemWithShares` class

```
1 class financialSystemWithShares(financialSystem):
2     def generate(self):
3     def generateQuantity(self):
4     def generateShares(self):
```

The method `generate()` is extended to include the generation of the `quantity` matrix, representing the amount of each type of asset that each bank holds, and the `shares` matrix, indicating the proportion of each bank's equity held by every other bank. Both `generateQuantity()` and `generateShares()` methods are called within the `generate()` method, which ensures that the financial system is set up with the appropriate ownership structure.

The `generateQuantity()` method is used to create the `quantity` matrix. The function generates this matrix by selecting the number of assets each bank owns from a normal distribution with the means as `quantityEdge` and standard deviation as $\sqrt{n}$. This process is repeated for each bank and each type of asset, resulting in an $n \times k$ matrix.

Similarly, the `generateShares()` function is used to generate the shares matrix. This function randomly selects some banks to be the owners of each bank's equity. The number of owners is drawn from a normal distribution with the mean as `sharesEdge` and standard deviation as $\sqrt{n}$. The shares are then distributed among the selected banks according to a uniform distribution that ranges from 0 and 1 and then normalized so that the total equity of each bank sums to 1.

3.2.1 Bankruptcy Cost

In the model proposed by Jackson and Pernoud, bankruptcy cost are also taken into account when a bank is unable to meet all its debts. In their paper, they mentioned that for the algorithm to successfully identify the best clearing payment vector, the function

$$d_i^A(V) - \beta_i(V,a) \tag{3.1}$$

must be non-decreasing in V. They provide an example of bankruptcy costs that fulfills the assumption:

$$\beta_i(V,a) = b + cA_i \tag{3.2}$$

where $b \geq 0,\ c \in [0,1],\ A_i = \sum_k q_{ik}a_k + \sum_j S_{ij}V_j^+ + d_i^A(V)$. This particular function is implemented within the `financialSystemWithShares()` class and is used as the bankruptcy cost in the financial system if `costFunction="linear"` is specified during initialization. The constants c and d are randomly generated following uniform distributions; c ranges from 0 to $0.01\bar{p}_i$ and d ranges from 0 to 1.

While a linear function is sufficient to evaluate the performance of the algorithms within different types of financial system, it is worth noting that this may not accurately reflect the real-world complexity, as bankruptcy cost often exhibit non-linear characteristics. Several studies have discovered that bankruptcy cost are a concave function of the firm's market value at the time of bankruptcy [14] [15] [16] [17]. For instance, Ang, Chua and McConnell estimated the costs and asset values using logarithmic and quadratic equations [18].

$$log\ \beta_i(V,a) = b_1 + c_1\ log\ A_i \tag{3.3}$$

$$\beta_i(V,a) = k + b_2A_i + c_2A_i^2 \tag{3.4}$$

In their paper, Ang, Chua and McConnell estimated the coefficients as $b_1 \approx 1.004,\ c_1 \approx 0.5359,\ k \approx 298,\ b_2 \approx 0.0286,\ c_2 \approx -0.0202 \times 10^{-6}$. These estimates are hypothesized from a random sample of 86 corporations spanning the period from 1963 to 1978 with

an estimated average liability and asset of 200,000 and 600,000. While the estimates may not be completely accurate today, given that the data are over 40 years old, they nonetheless provide valuable reference points.

If the `costFunction` parameter is set to ”log”, then equation 3.3 is employed. Conversely, when `costFunction` is set to ”quadratic”, then equation 3.4 is used. When using the logarithmic function to calculate bankruptcy costs, equation 3.1 is a decreasing function within the range of $0 < V < x$, where the specific value of x depends on the selected coefficients for the logarithmic function. To ensure that equation 3.1 remains non-decreasing in V, a modification is applied to the bankruptcy cost when the logarithmic function is used. Specifically, a minimum amount of bankruptcy cost is incurred when $0 < V < x$.

On the other hand, the quadratic function is a decreasing function when $V > \frac{b_2}{-2c_2}$. Hence, in the implementation, the value vector used to calculate the bankruptcy cost will be upper bounded by $\frac{b_2}{-2c_2}$. The value vector used in other parts of the algorithms will remain unchanged.

3.2.2 Implementation of the Algorithm by Jackson and Pernoud

```
1 class financialSystemWithShares(financialSystem):
2     def solve(self):
```

The algorithm is implemented within the `solve()` function of the `financialSystemWithShares()` class. This algorithm diverges from the other algorithms as it calculates the best clearing payment matrix, rather than a vector at its conclusion. Each (i, j) entry of the matrix corresponds to the best clearing payment value from bank i to bank j.

Initially, the matrix is configured to match the `liabilityMatrix` while the value of each bank, represented by `valueVector`, is set to 0. Additionally, all banks are assumed solvent at the onset, thus `solventMatrix` is equivalent to an `n`-by-`n` identity matrix.

The algorithm starts an iteration process, persisting until `valueVector` converges to a fixed point. During each iteration, the algorithm computes the total cash of each bank, encompassing both primitive and debt assets. This information is utilised to determine the solvency of a bank, considering its total cash and total debt payments. Subsequently, the bankruptcy costs for defaulting banks are computed and recorded in an array called

bankruptcyCost.

The payment value of defaulting banks is then updated in accordance with equation 2.5, while the value of all banks is updated per equation 2.6. As mentioned in Section 3.1.1, the implementation leverages of NumPy's built-in functions such as where, maximum, dot, sum and more to enhance computational efficiency.

```
class financialSystemWithShares(financialSystem):
    def calculateTotalCash(self):
    def updateSolventMatrix(self, totalCash):
    def updateBankruptcyCost(self, totalCash):
```

Within the solve() function, several helper functions are utilised to improve code readability and description. The calculateTotalCash() function is first employed at the start of each solve() iteration, determining the total cash for all banks. Next, updateSolventMatrix() is invoked during each iteration to update the solvent Matrix. Lastly, the bankruptcy costs for defaulting banks are calculated using the updateBankruptcyCost() function. The specific function used to compute the cost depends the costFunction parameter, as described in Section 3.2.1.

3.3 Proof of Correctness

As mentioned in Section 3.1.1, the close() function is utilised to confirm the convergence of the clearing payment vector to a fixed point. This function calculates the Euclidean distance between the last and successive payment vectors and returns True if the Euclidean distance is less than maxDistance.

However, if maxDistance is not adequately small, the computed clearing payment vector may significantly deviate from the best clearing payment vector. To ensure the accuracy of the computed clearing payment vector from an algorithm, it can be cross-verified with the solution derived from the linear programming solver. The linear programming solver provides a high degree of accuracy as it does not rely on the close() function. It is worth noting that the other algorithms can achieve greate accuracy than the linear programming solver if maxDistance is set to an extremely small value, such as zero. However, this could potentially slow down the execution of the algorithms.

```
class financialSystem():
    def validPyament(self, payment, output=False):
```

```
3
4 class financialSystemWithShares(financialSystem):
5     def validPayment(self, payment, output=False):
```

We can use the same method to verify the accuracy of the algorithm by Jackson and Pernoud when $shares_{ij} = 0$ for all $i > 0,\ j > 0$. In other cases, the correctness of the clearing payment vector can be assessed by calculating the total cash (or assets) of the banks. This assumes that all banks fulfill their debts in accordance with the computed clearing payment vector. Subsequently, we need to ensure that their total cash is greater than or approximately equal to (as determined by the `close()` function) their clearing payments. This validation technique is implemented in the `validPayment()` function, which confirms that the computed clearing payment vector is indeed accurate. However, it should be noted that this method does not ascertain that the computed vector is the best clearing payment vector.

```
1 class financialSystem():
2     def bestPayment(self):
3
4 class financialSystemWithShares(financialSystem):
5     def bestPayment(self):
```

To ascertain the correctness of the computed clearing payment vector, a small value can be added to each entry of the clearing payment vector. This process is executed within a function named `bestPayment()`. If the vector is truly the best clearing payment vector, then `validPayment()` should return `False` after this addition. In this implementation, the added value is determined to be `2 * maxDistance`/$\sqrt{n}$. This strategy serves as a check to ensure that the computed vector is not only correct but also optimal.

Chapter 4

Experimental Evaluation of the Implementation

In this chapter, a series of experiments will be conducted to evaluate the performance of the algorithms under various conditions. Throughout these experiments, the computed clearing payment vector will be verified for correctness and optimality using the methods mentioned in Section 3.3. Unless otherwise specified, the `maxDistance` will be set as 0.0001 and the `costFunction` will be configured as `"logarithmic"`. Additionally, all results presented are derived from the average of ten separate trial runs to minimise variability.

While the algorithm by Dang, Qi and Ye has been implemented, it will not be included in the experiments, with the exception of Section 4.4. This decision is based on preliminary results showing that this algorithm consistently underperforms compared to the other algorithms in various scenarios, as demonstrated in Section 4.4. This might be due to the fact that the algorithm operates by identifying the midpoint of the search lattice, which is a different approach compared to the other iterative algorithm that may fortuitously converge towards the best clearing payment vector.

Another potential factor could be that the implementation of the Dang, Qi and Ye algorithm might not be sufficiently efficient. It should be noted, however, that we have taken every possible step to ensure that the implementation is as efficient as possible, mirroring the optimisation efforts applied to the other algorithm implementations.

In this chapter, abbreviations will be used in place of full algorithm names for the sake of brevity. The following abbreviations correspond to the algorithms as described:

- BIA(top) and BIA(bottom) represent the Basic Iteration Algorithm, with (top) and (bottom) indicating whether the initial clearing payment vector is set as $\bar{p}$ or

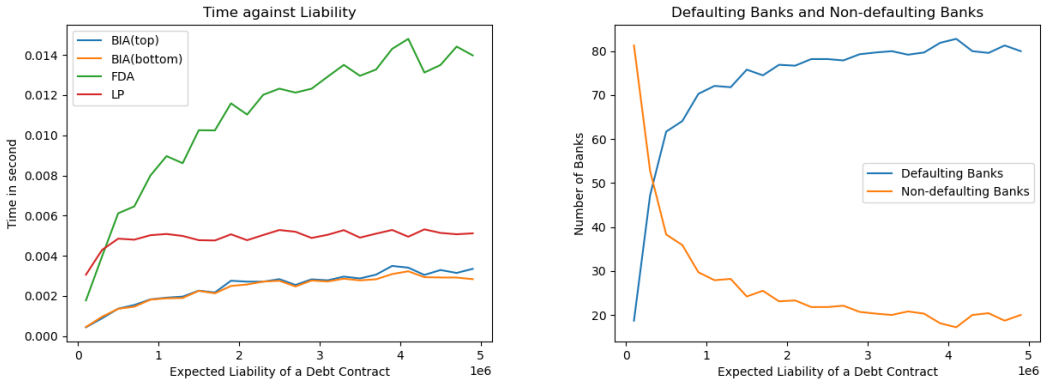
$\bar{0}$, respectively.

- FDA stands for the Fictitious Default Algorithm.
- LP denotes the algorithm that utilises Linear Programming.
- JP is the abbreviation for the algorithm proposed by Jackson and Pernoud.

4.1 Bankruptcy Rate

In this section, we conduct experiments by manipulating the bankruptcy rate within the financial system, defined as the ratio of defaulting banks to non-defaulting banks when all banks settle their debts according to the best clearing payment vector. Primarily, three parameters can influence this bankruptcy rate: `liability`, `asset`, and `edge`. Our investigation will proceed sequentially, examining the effects of altering each parameter, beginning with `liability`, followed by `asset`, and finally, the `edge` parameter.

4.1.1 `liability`



(a) Time against `liability` (b) Number of Banks against `liability`

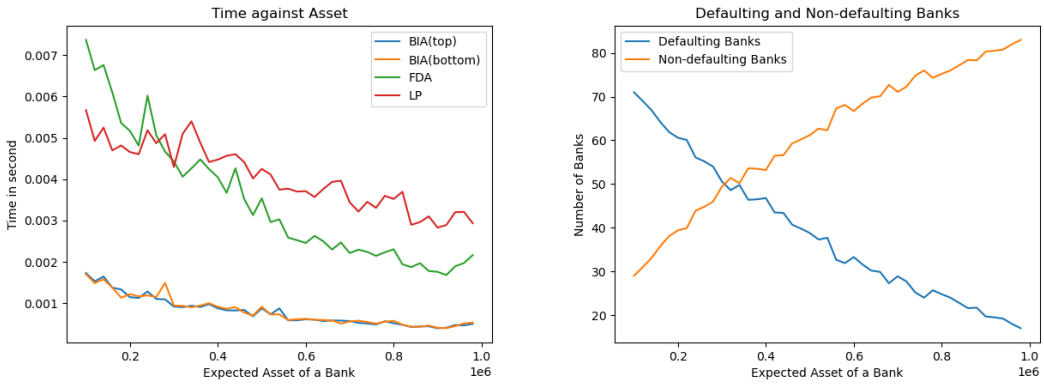
Figure 4.1

Figure 4.1 presents the results of experiments conducted with parameters set as `n=100`, `asset=1e6`, `edge=10` and `liability` ranging from 1e5 to 5e6. As shown in Figure 4.1a, the performance ranking of the algorithms, from the fastest to slowest, is as follows: BIA(top), BIA(bottom), FDA and LP. However, as the expected liability of a debt contract increases, this order shifts to BIA(top), BIA(bottom), LP and FDA.

This change in performance ranking can be explained by Figure 4.1b. As the expected liability of a debt contract increases, so too does the number of defaulting banks. In other words, the bankruptcy rate within the financial system escalates with an increase in the expected liability of a debt contract. As mentioned in Chapter 3, the iterative algorithms will set the clearing payment of a bank as the total amount of its debt in a single iteration if the bank has sufficient funds to cover it. Therefore, the iterative algorithms outperform the linear programming solver when the bankruptcy rate is low. The performance of the iterative algorithms decreases as the bankruptcy rate increases, which is especially true for the FDA algorithm. This is due to the fact that more iterations are needed to calculate the best clearing payment vector as the vector itself is getting larger.

It is noteworthy that the BIA(bottom) algorithm initially performs comparably to the BIA(top) algorithm, but its performance deteriorates more as `liability` increases. This can be attributed to the fact that the best clearing payment vector is generally closer to $\bar{p}$ than $\bar{0}$. While it may seem premature to assert that BIA(bottom) underperforms BIA(top), further sections of this chapter will provide evidence supporting this claim.

4.1.2 `asset`



(a) Time against `asset` (b) Number of Banks against `asset`

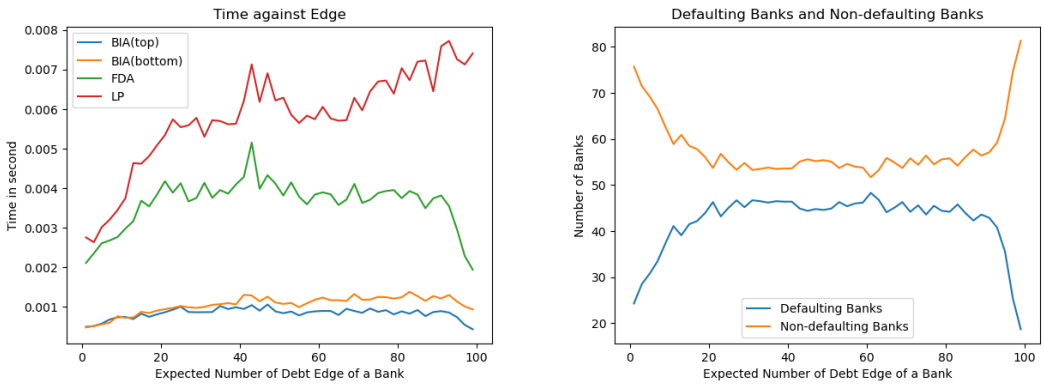
Figure 4.2

Figure 4.2 depicts the performance of financial systems characterised by the parameters: `n=100`, `liability=1e5`, `edge=10` and `asset` varying from 1e5 to 1e6. Given that modifying the `asset` parameter essentially inversely correlates with altering `liability`, the trends observed in Figure 4.2a are quite anticipated.

One notable point, however, is the smaller impact that `asset` adjustments have on the bankruptcy rate compared to `liability` changes, as evident in Figure 4.2b and Figure 4.1b. This can be attributed to the `edge` parameter set at 10, which means that changes in `liability` can have a more pronounced effect on a bank's total cash flow.

Interestingly, the BIA(bottom) algorithm's performance parallels that of the BIA(top) algorithm in these scenarios. As previously discussed, the iterative algorithms typically require only a single iteration to compute the best clearing payment value for solvent banks. Therefore, as the number of non-defaulting banks increases with an increase in `asset`, the performances of BIA(bottom) and BIA(top) algorithms converge. Meanwhile, FDA initially is slower than the LP algorithm but this changes as the bankruptcy rate of the financial system decreases.

4.1.3 edge



(a) Time against `edge` (b) Number of Banks against `edge`

Figure 4.3

Figure 4.3 illustrates the performance of financial systems characterised by the parameters: `n = 100`, `liability=2e5`, `asset=1e6` and `edge` varying from 1 to 100. As shown in Figure 4.3a and Figure 4.3b, the performance of the iterative algorithms fluctuates in accordance with changes in the bankruptcy rate.

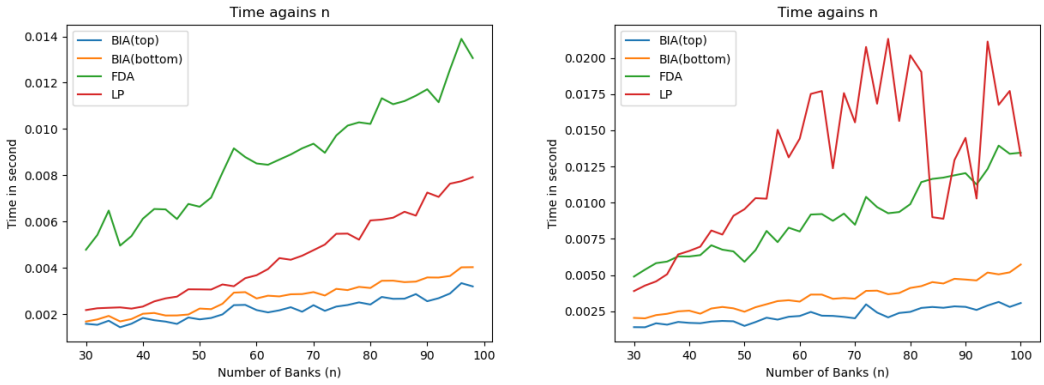
Contrarily, the LP algorithm's performance decreases as the `edge` parameter increases, even when the bankruptcy rate is low when `edge` is near 100. This suggests that the LP algorithm's performance is more reliant on the complexity of the financial system than on the bankruptcy rate. We define the complexity of a financial system as the total number of directed edges between banks.

An intriguing observation is that all the iterative algorithms, with the exception of BIA(bottom), seem to be less influenced by the complexity of the financial system and more influenced by the bankruptcy rate. However, the performance of BIA(bottom) does fluctuate with the bankruptcy rate, similar to other iterative algorithms, but it also displays a higher sensitivity to the complexity of the financial system compared to BIA(top).

4.2 Complexity of the Financial System

In this section, experiments will be conducted by changing the complexity of the financial system, which is influenced by two parameters: `n` and `edge`. These experiments will demonstrate that the performance of the LP algorithm is significantly affected by the complexity of the financial system. However, this observation does not necessarily imply that the LP algorithm performs the worst among all algorithms when dealing with complex financial systems.

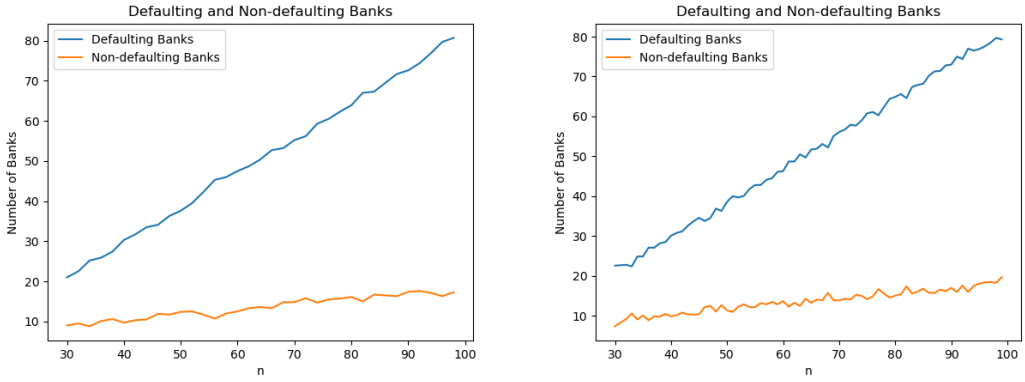
4.2.1 Effect of Varying Complexity on LP



(a) `edge` = 0.5n (b) `edge` = n

Figure 4.4: Effect of Complexity 1

In Section 4.1.3, we observed that the performance of the LP algorithm is highly dependent on the complexity of the financial system, especially when compared to other algorithms. Figure 4.4 provides further evidence to support this claim. Note that when the expected debt edge of a bank is higher, the performance of the LP algorithm is worse than that of the other algorithms, as seen when comparing Figure 4.4a with Figure 4.4b.

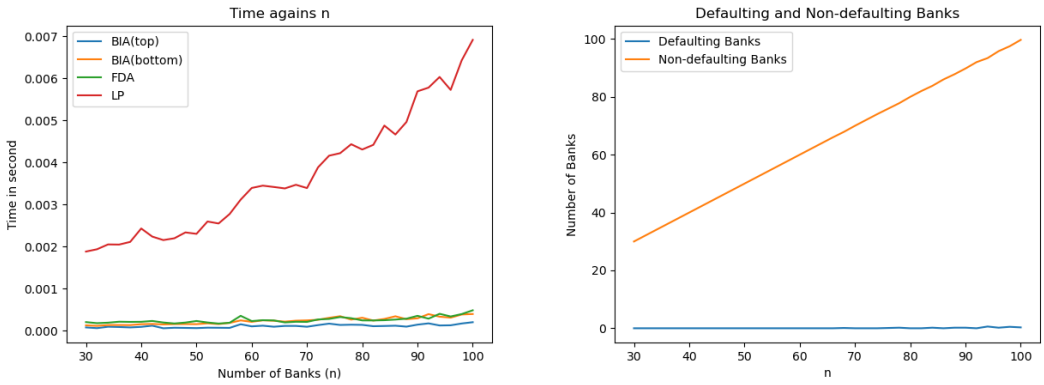


(a) `edge` = 0.5n (b) `edge` = n

Figure 4.5: Effect of Complexity 2

This change in trend is not attributable to the bankruptcy rate of the financial system. As Figure 4.5 illustrates, the bankruptcy rates of the financial systems in both scenarios are approximately the same. This is due to the nature of linear programming, a more complex linear programming problem requires more time to be solved by a linear programming solver. The experiments were conducted with the following parameters: `asset=1e5`, `liability=1e5`, `n` ranges from 30 to 100 and `edge` is either `0.5n` or `n` in Figure 4.4a or Figure 4.4b, respectively.

4.2.2 When Banks are Mostly Solvent



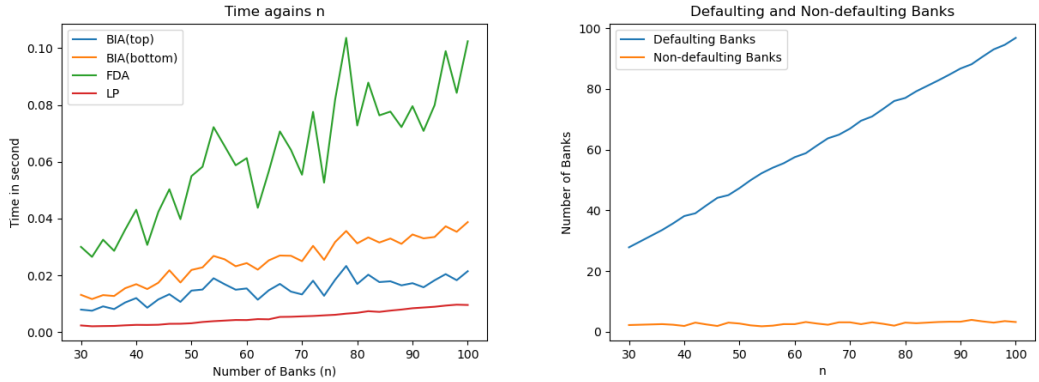
(a) (b)

Figure 4.6: Mostly Solvent

In Section 4.1, we noted that the iterative algorithms can swiftly determine the best payment values when a bank is solvent. Figure 4.6 further evidence to give

support to this observation. A significant gap can be seen between the performance of the LP algorithm and that of the other algorithms when all the banks are solvent. The experiments depicted in Figure 4.6 were conducted with the following parameter settings: `asset=1e6`, `liability=1e5`, `edge=n` and `n` ranges from 30 to 100.

4.2.3 When Banks are Mostly Defaulting



(a) (b)

Figure 4.7: Mostly Defaulting

Section 4.2.2 does not suggest that the iterative algorithms will always outperform the LP algorithm. As demonstrated in Figure 4.8, the LP algorithm emerges as the fastest when the bankruptcy rate is high. As mentioned before, the iterative algorithms require more iterations to identify the best clearing payment vector as the best clearing payment vector gets larger. The parameters used in this scenario include `asset=1e5`, `liability=1e6`, `edge=n` and `n` ranges from 30 to 100.

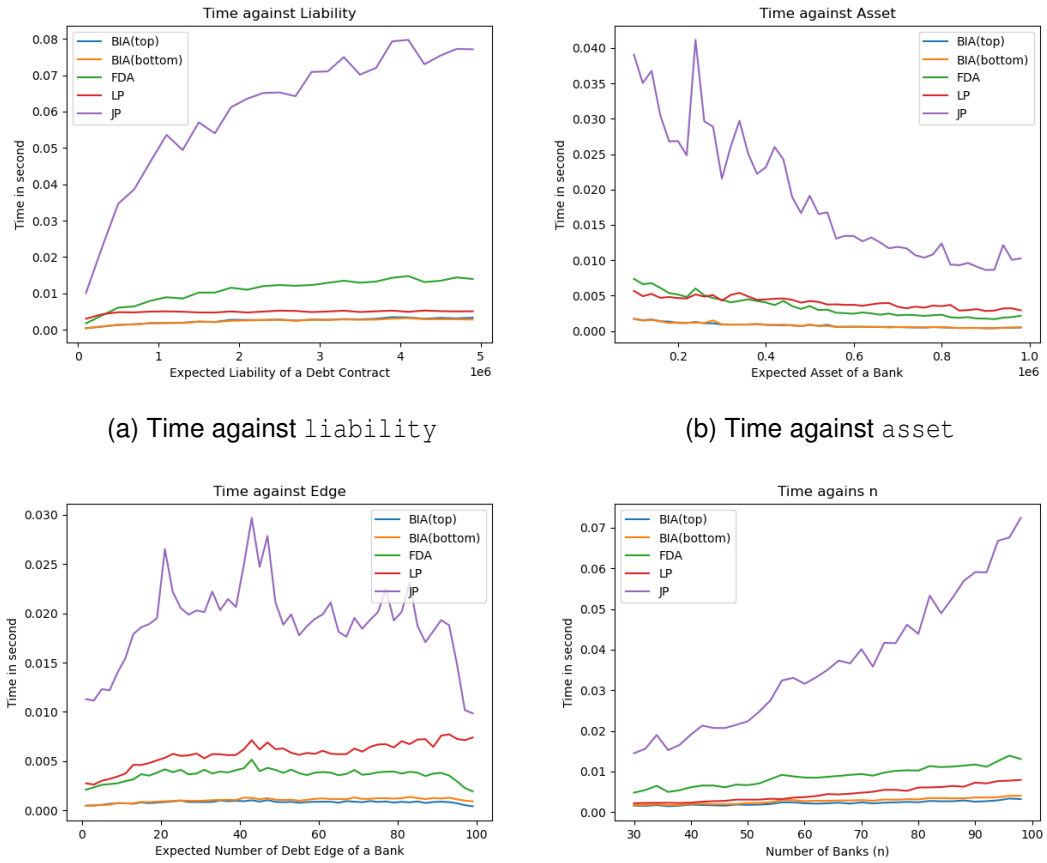
4.3 The Algorithm by Jackson and Pernoud

In this section, we explore experiments tailored to the algorithm by Jackson and Pernoud. Specifically, we investigate how the algorithm's performance is influenced by `sharesEdge` and `quantityEdge`. Also, the algorithm tends to underperform other algorithms in the model by Eisenberg and Noe, where factors like equity shares and bankruptcy costs are not incorporated. We conduct the same experiments as detailed in Section 4.1 and Section 4.2 to include the JP algorithm, with the results displayed in Figure 4.8 and Figure 4.9.

4.3.1 Comparing with the Other Algorithms

The reason why the JP algorithm generally underperforms compared to other algorithms in the same setting is due to its method of operation. Unlike other iterative algorithms that only update the clearing payment vector in each iteration, the JP algorithm updates both the clearing payment matrix and the value vector of banks during every cycle.

This additional computation in every iteration causes the JP algorithm to perform less efficiently than the other algorithms. However, it is worth noting that the JP algorithm has the advantage of calculating the clearing payment matrix instead of just the vector. This allows for the determination of clearing payments between individual banks.



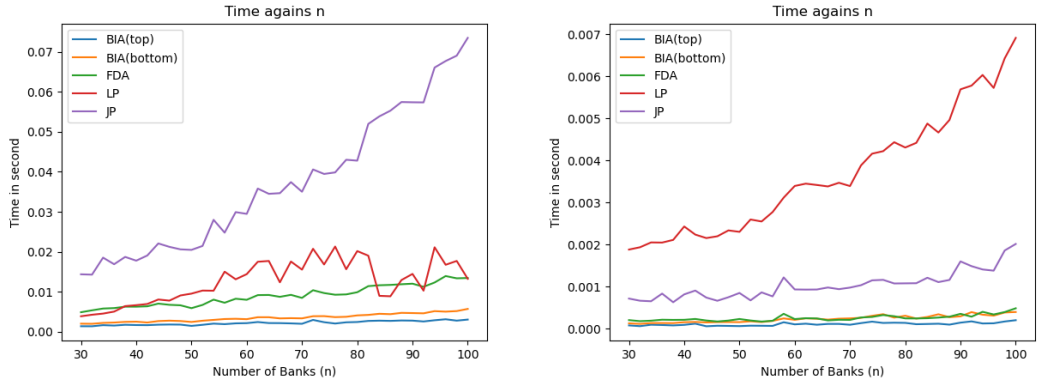
(a) Time against `liability`

(b) Time against `asset`

(c) Time against `edge`

(d) Time against `n`

Figure 4.8: The Performance of the Algorithm by Jackson and Pernoud against Other Algorithms 1

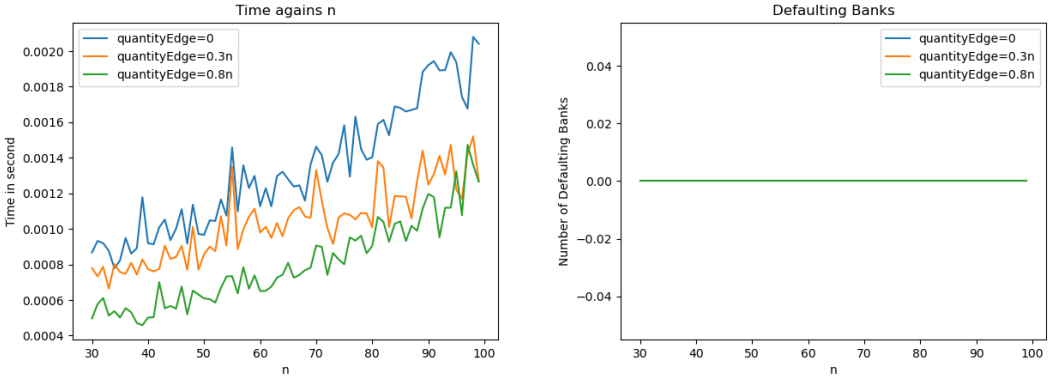


(a) High Bankruptcy Rate (b) Low Bankruptcy Rate

Figure 4.9: The Performance of the Algorithm by Jackson and Pernoud against Other Algorithms 2

4.3.2 Multiple Assets and Equity Shares

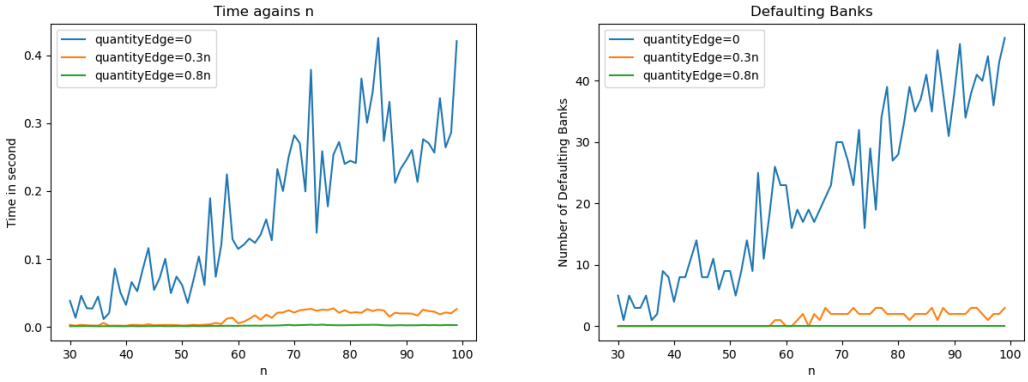
Owning k different assets, where each asset i has a value of a_i, is the same as having a single asset with a combined value of $\sum_k a_i$. Thus, increasing the `quantityEdge` parameter does not substantially alter the number of iterations needed to find the best clearing payment matrix if the expected total asset of a bank is consistent.



(a) (b)

Figure 4.10: `quantityEdge` (Same Bankruptcy Rate)

However, increasing `quantityEdge` extends the time required to calculate the total cash of each bank in every iteration of the algorithm. Consequently, the performance of the algorithm marginally decreases as `quantityEdge` increases if the bankruptcy rate is kept constant, as shown in Figure 4.10, under the following parameters: `asset=1e6`, `liability=1e5`, `debtEdge=50`, `costFunction="none"`, `quantityEdge=[0, 0.3n`

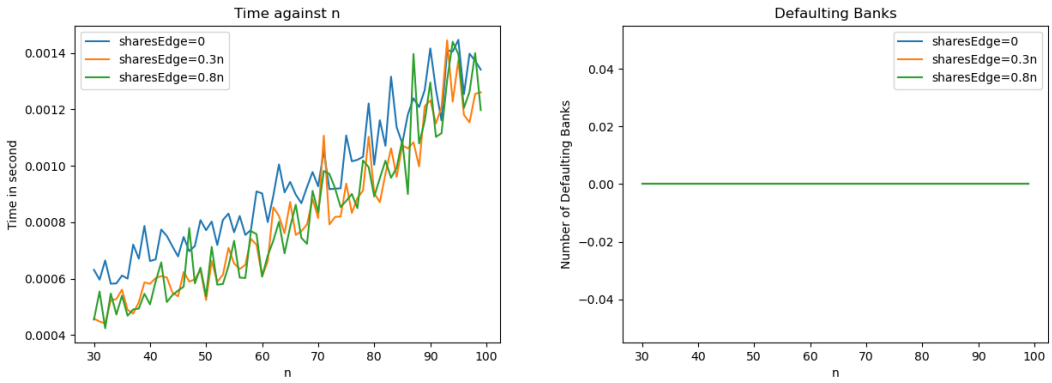


(a) (b)

Figure 4.11: `quantityEdge` (Different Bankruptcy Rate)

`, 0.8n]`, `sharesEdge=30`, `k = 100` and `n` ranges from 30 to 100.

Additionally, an increase in `quantityEdge` can change the bankruptcy rate of the financial system if the other parameters are kept constant, thereby affecting the algorithm's performance. This effect is illustrated in Figure 4.11, where the parameters are the same as in Figure 4.11, except that `liability` is now set at `1e7`.

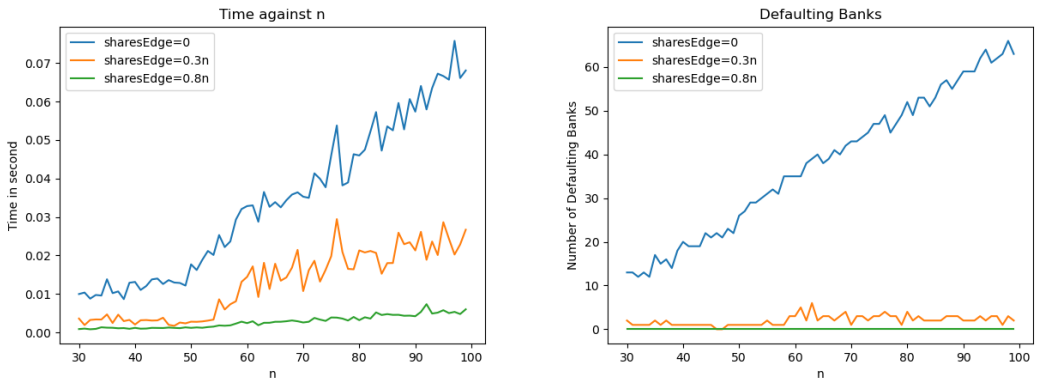


(a) (b)

Figure 4.12: `sharesEdge` (Same Bankruptcy Rate)

The concept can also be applied to `sharesEdge`. Increasing `sharesEdge` will decrease the time needed to calculate the best clearing payment matrix due to the same reason used to explain the effect of `quantityEdge`. In this case, equity shares of other banks can be considered as an additional asset owned by the shares holder. Figure 4.12 shows that the performance of the algorithm decreases by a little bit when `sharesEdge` increases even when the bankruptcy rate does not change. The param-

eters for the scenario include: `asset=1e6`, `liability=1e5` `costFuntion="none"`, `quantityEdge=1`, `sharesEdge=[0, 0.3n, 0.8n]`, `k = 100` and `n` ranges from 30 to 100. Figure 4.13 shows that the performance of the algorithm increases due to the significant drop in bankruptcy rate when `sharesEdge` increases.

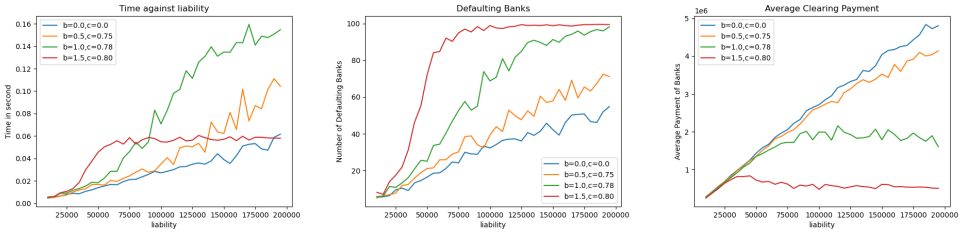


(a) (b)

Figure 4.13: `sharesEdge` (Different Bankruptcy Rate)

4.3.3 Bankruptcy Cost

As detailed in Section 3.2.1, three types of functions are implemented to represent bankruptcy costs: linear, logarithmic and quadratic. Each type of bankruptcy cost is examined individually to investigate its effect when the coefficients of the functions are increased.

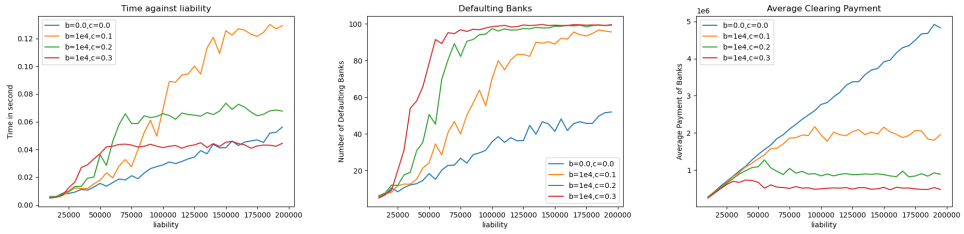


(a) (b) (c)

Figure 4.14: Logarithmic Cost Function

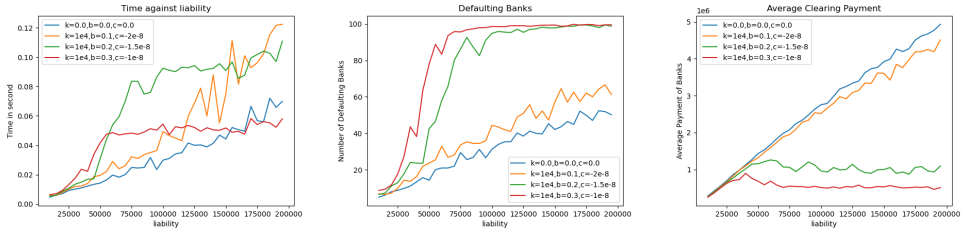
Figure 4.14 presents the results of experiments with the following parameters: `n = 11`, `k = 100`, `quantityEdge=1`, `sharesEdge=30`, `debtEdge=50`, `liability` ranges from `1e4` to `2e5`. Equation 3.3 is utilised as the bankruptcy cost in these experiments.

From Figure 4.14a, it is evident that increasing the bankruptcy cost by augmenting the coefficients leads to a decrease in the performance of the algorithm, except when $b = 1.5, c = 0.8$. This drop in performance can be explained by Figure 4.14b, while the exception can be explained by Figure 4.14c. A rise in bankruptcy cost increases the bankruptcy rate, hindering the algorithm's performance. However, if the bankruptcy rate is excessively large, the total cash flow in the financial system diminishes. This allows the algorithm to find the best clearing payment matrix with fewer iterations as the average of the best clearing payment matrix is now smaller, as shown in Figure 4.14c.



(a) (b) (c)

Figure 4.15: Linear Cost Function



(a) (b) (c)

Figure 4.16: Quadratic Cost Function

Figure 4.15 and Figure 4.16 are the results of experiments when the bankruptcy cost functions are set as linear and quadratic respectively. From Figure 4.15, it can be observed that the time required for the financial systems with $c = 0.2$ and $c = 0.3$ is upper-bounded by 0.07 and 0.04 seconds respectively. This pattern, as explained when using the logarithmic cost function, is clarified in Figure 4.15c. Figure 4.15b further demonstrates that the bankruptcy rate of the financial systems is heavily influenced by the bankruptcy cost.

A similar trend is visible with the quadratic cost functions, as shown in Figure 4.16. The distinction between linear and quadratic cost functions lies in the increased upper

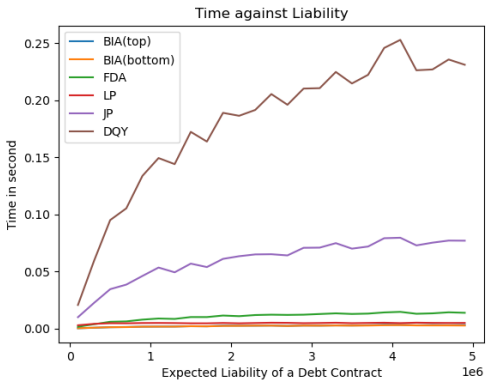
bound on the time required, a consequence of the quadratic factors in the cost function. The parameters of the financial systems in both figures align with those of the financial systems employing logarithmic cost functions.

4.4 The Algorithm by Dang, Qi and Ye

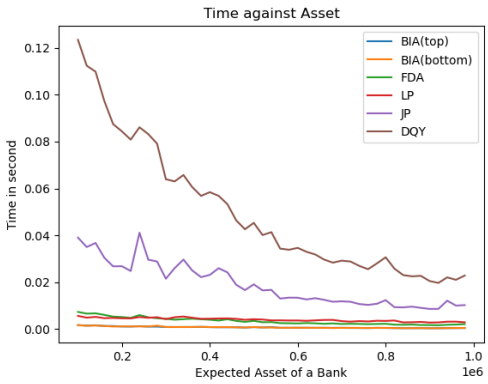
In this section, we present the results of experiments from Section 4.1 and Section 4.2 that includes the DQY algorithm. As shown in Figure 4.17, DQY underperforms all other algorithms in nearly every scenario, with the exception of cases where the financial systems comprise mostly solvent banks. This exception occurs because the iteration algorithms only require a single iteration to identify the best clearing payment for solvent banks, giving them an advantage.

The subpar performance of DQY may be attributed to the nature of the underlying monotone function (Equation 2.1) used to compute the best clearing payment vector. This function may converge to a fixed point rapidly, but the performance of DQY does not depend on this quick convergence. Instead, DQY operates by locating the midpoint of the lattice which represents all potential values of the best clearing payment vector. Then, the algorithm employs a divide-and-conquer strategy to recursively find the best clearing payment vector within a multi-dimensional lattice.

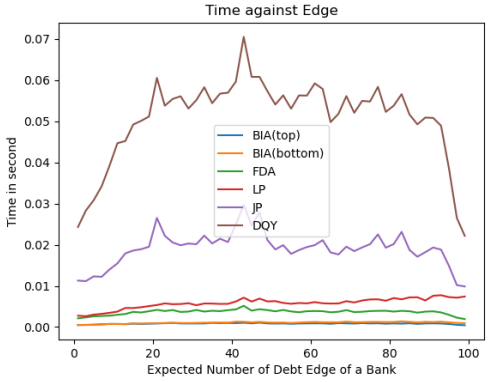
Interestingly, DQY outperforms all other algorithms and aligns closely with BIA(top) in scenarios where banks are mostly solvent. However, as shown in Figure 4.17f, DQY's performance diminishes sharply only after `n` increases past 75. This observation implies that the algorithm could potentially compete with others if the convergence rate of the monotone function were artificially controlled. Nevertheless, we did not conduct such an experiment, as it would be unrealistic to artificially slow down the performance of the other algorithms to match DQY's performance.



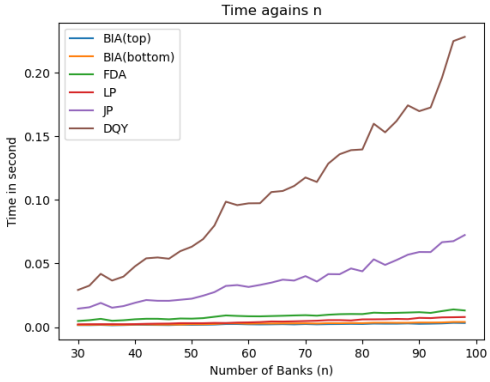
(a) Time against `liability`



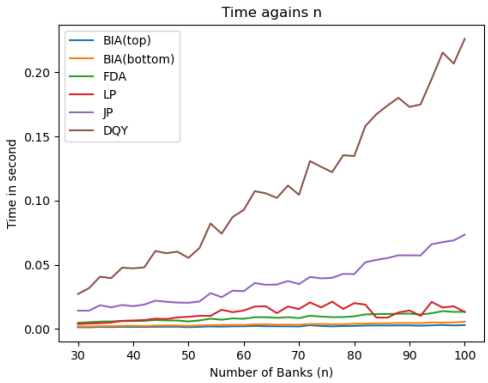
(b) Time against `asset`



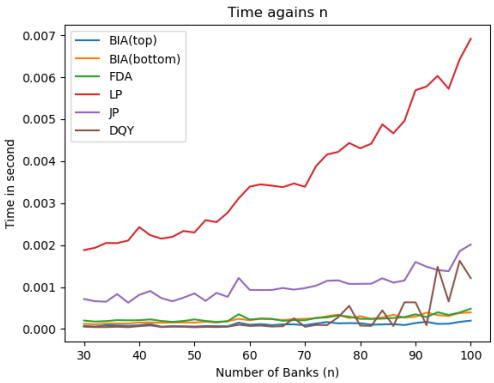
(c) Time against `edge`



(d) Time against `n`



(e) High Bankruptcy Rate



(f) Low Bankruptcy Rate

Figure 4.17: The Performance of the ALgorithm by Dang, Qi and Ye against Other Algorithms

Chapter 5

Conclusion

The paper by Eisenberg and Noe introduces a model that involves direct debt contracts between banks, with each bank also possessing an operating cash flow or asset. In this model, the authors present three distinct algorithms designed to identify the optimal clearing payment vector. Conversely, the model proposed by M. O. Jackson and A. Pernoud builds upon the previous framework by permitting banks to hold shares in other banks. Furthermore, each bank is not limited to a single operating cash flow; instead, a variety of assets exist, with different banks investing in these assets in varying quantities. In this expanded model, the authors outline an algorithm that can be employed to discover the best clearing payment vector.

In this study, the model defined by Eisenberg and Noe is implemented and evaluated. The algorithms detailed in the same paper are executed, and experiments are conducted to assess these algorithms in diverse financial contexts. The bankruptcy rate of a financial system is defined as the ratio of defaulting banks to solvent banks when all banks pay their debt according to the optimal clearing payment vector. This study demonstrates that the algorithm utilising linear programming is the fastest when the bankruptcy rate is high. However, it is also observed that this linear programming algorithm's performance is significantly affected by the financial system's complexity, rendering it slower than other algorithms in certain scenarios. Conversely, the basic iteration algorithm significantly outperforms other algorithms in most cases.

Eisenberg and Noe prove that, under specific conditions, their model's clearing payment vector is unique. Thus, as long as these conditions are fulfilled, any other algorithm designed to find a fixed point of a monotone function can be applied to this problem. An algorithm by Dang, Qi, and Ye, aimed at finding any fixed point of a monotone function, is also incorporated into this study for comparison. Although it

boasts a faster theoretical complexity than the basic iteration algorithm, the research shows that it underperforms other algorithms most of the time.

The extended model by M. O. Jackson and A. Pernoud is also implemented and tested in this study. While faster than the algorithm by Dang, Qi, and Ye, it is slower than other algorithms most of the time. However, the algorithm by Jackson and Pernoud possesses the unique advantage of being able to compute the best clearing payment matrix, detailing the clearing payments between banks, rather than merely a vector.

When factors such as equity shares and bankruptcy costs are introduced into the financial system, this study reveals that the bankruptcy rate can be influenced by these parameters. Specifically, possessing more equity shares in other banks decreases the bankruptcy rate, while higher bankruptcy costs increase it. In scenarios where the bankruptcy rate remains constant, having more equity shares and multiple assets can actually reduce the performance of the algorithm.

5.1 Future Work

One area ripe for exploration is the application of these algorithms to credit default swaps (CDS). This complex financial instrument plays a crucial role in modern finance, and the algorithms studied here could provide innovative methods for modelling, analyzing, and managing CDS risk. Investigating the algorithms' applicability to credit default swaps could open new avenues for research and practical implementation in financial risk management.

Beyond credit default swaps, future work could also delve into other sophisticated financial products and markets, such as options pricing, portfolio optimization, and systemic risk assessment. These areas could benefit from the refinement of existing algorithms or the development of entirely new computational methods tailored to their unique challenges.

Additionally, the research could lead to the exploration of new financial contexts and the development of more robust evaluation techniques. Efforts to bridge theoretical constructs with practical applications would enhance the real-world relevance of the algorithms. Collaborations with financial institutions and regulatory bodies might offer opportunities to test the algorithms in real market conditions, providing further validation of their efficacy and adaptability.

Bibliography

[1] Larry Eisenberg and Thomas H. Noe. Systemic risk in financial systems. *Management Science*, 47(2):236–249, 2001.

[2] Matthew O. Jackson and Agathe Pernoud. What makes financial markets special? systemic risk and its measurement in financial networks. *SSRN Electronic Journal*, 2019.

[3] Chong Shu. Endogenous risk-exposure and systemic instability. *SSRN Electronic Journal*, 2018.

[4] Adam Zawadowski. Entangled Financial Systems. *The Review of Financial Studies*, 26(5):1291–1323, 03 2013.

[5] Andrea Galeotti and Christian Ghiglino. Cross-ownership and portfolio choice. *Journal of Economic Theory*, 192:105194, 2021.

[6] Rakesh Vohra, Yiqing Xing, and Wu Zhu. The network effects of agency conflicts. *SSRN Electronic Journal*, 2020.

[7] Steffen Schuldenzucker, Sven Seuken, and Stefano Battiston. Finding Clearing Payments in Financial Networks with Credit Default Swaps is PPAD-complete. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*, volume 67 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:20, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] Stavros D. Ioannidis, Bart de Keijzer, and Carmine Ventre. Strong approximations and irrationality in financial networks with financial derivatives, 2021.

[9] Chuangyin Dang, Qi Qi, and Yinyu Ye. Computations and complexities of tarski's fixed points and supermodular games, 2020.

[10] John Fearnley, Dömötör Pálvölgyi, and Rahul Savani. A faster algorithm for finding tarski fixed points, 2020.

[11] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.

[12] I. Stančin and A. Jović. An overview and comparison of free python libraries for data mining and big data analysis. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 977–982, 2019.

[13] Minsuan Teh. Algorithms for computing a tarski fixed point on a finite lattice, equilibria in submodular games and games with strategic complementarities, 2022.

[14] T. Randolph Beard. Bankruptcy and care choice. *The RAND Journal of Economics*, 21(4):626, 1990.

[15] Donald R. Deis, Daryl M. Guffey, and William T. Moore. Further evidence on the relationship between bankruptcy costs and firm size. *Quarterly Journal of Business and Economics*, 34(1):69–79, 1995.

[16] Michael E Bradbury and Suzanne Lloyd. An estimate of the direct costs of bankruptcy in new zealand. *Asia Pacific Journal of Management*, 11(1):103–111, April 1994.

[17] Jerold B. Warner. Bankruptcy costs: Some evidence. *The Journal of Finance*, 32(2):337, May 1977.

[18] JAMES S. ANG, JESS H. CHUA, and JOHN J. MCCONNELL. The administrative costs of corporate bankruptcy: A note. *The Journal of Finance*, 37(1):219–226, March 1982.

Appendix A

Python Implementaion Code

A.1 financialSystem.py

```
import numpy as np
from scipy.optimize import linprog
import timeit

class financialSystem():
    def __init__(self, n, liability, asset, maxDistance=1, edge=0):
        self.n                          = n
        self.clearingPaymentVector      = np.zeros(n)
        self.assetVector                = np.zeros(n)
        self.liabilityMatrix            = np.zeros((self.n, self.n))
        self.maxDistance                = maxDistance
        self.edge                       = edge
        self.asset                      = asset
        self.liability                  = liability
        self.relativeLiabilityMatrix    = np.zeros((self.n, self.n))
        self.totalPaymentVector         = np.zeros(self.n)


    def generate(self):
        if self.n < 2:
            print("n must be greater than 1")
            return

        self.assetVector    = np.random.normal(loc=self.asset,
        ↪  scale=np.sqrt(self.n), size=self.n).astype(np.float64)
        self.edge           = max(1, min(self.edge, self.n - 1))

        self.liabilityMatrix                    = np.zeros((self.n, self.n))
        mu                                      = self.edge
```

```
29          sigma                                   = np.sqrt(self.n)
30
31          seed                                    = np.random.normal(mu, sigma,
            ↪  size=self.n)
32          seed                                    = np.maximum(np.zeros(self.n),
            ↪  np.minimum(np.round(seed), np.ones(self.n) * self.n - 1))
33
34          for i in range(self.n):
35              position                            = np.random.choice(self.n - 1,
                ↪  replace=False, size=int(seed[i]))
36              position[np.where(position >= i)]   += 1
37              self.liabilityMatrix[i, position]   =
                ↪  np.random.normal(loc=self.liability, scale=np.sqrt(self.n),
                ↪  size=int(seed[i])).astype(np.float64)
38
39          self.totalPaymentVector         = np.sum(self.liabilityMatrix, axis=1)
40          for i in range(np.size(self.totalPaymentVector)):
41              if self.totalPaymentVector[i] == 0:
42                  self.totalPaymentVector[i] = -1
43          self.relativeLiabilityMatrix    = self.liabilityMatrix /
            ↪  self.totalPaymentVector[:, None]
44          for i in range(np.size(self.totalPaymentVector)):
45              if self.totalPaymentVector[i] == -1:
46                  self.totalPaymentVector[i] = 0
47
48      def solve(self, topToBottom=True):
49          self.clearingPaymentVector  = np.zeros(self.n)
50          if topToBottom:
51              self.clearingPaymentVector  = np.array(self.totalPaymentVector)
52          same                        = False
53          while not same:
54
55              oldPaymentVector            = np.array(self.clearingPaymentVector)
56              newPaymentVector            = np.array(self.assetVector)
57              newPaymentVector            +=
                ↪  np.dot(np.transpose(self.relativeLiabilityMatrix),
                ↪  self.clearingPaymentVector)
58              self.clearingPaymentVector  = np.minimum(newPaymentVector,
                ↪  self.totalPaymentVector)
59              same                        = self.close(oldPaymentVector,
                ↪  self.clearingPaymentVector)
60          return self.clearingPaymentVector
61
62      def solveShowGraph(self):
63          self.clearingPaymentVector  = np.zeros(self.n)
64          same                        = False
```

```
65          listOfDistance              = []
66          while not same:
67
68              oldPaymentVector            = self.clearingPaymentVector
69              newPaymentVector            = np.zeros(self.n) + self.assetVector
70              newPaymentVector            +=
                ↪  np.dot(np.transpose(self.relativeLiabilityMatrix),
                ↪  self.clearingPaymentVector)
71              self.clearingPaymentVector  = np.minimum(newPaymentVector,
                ↪  self.totalPaymentVector)
72              same                        = self.close(oldPaymentVector,
                ↪  self.clearingPaymentVector)
73
74              distance = np.sqrt(np.sum((oldPaymentVector -
                ↪  self.clearingPaymentVector)**2))
75              listOfDistance.append(distance)
76          return self.clearingPaymentVector, listOfDistance
77
78      def close(self, arrayA, arrayB, start=None, end=None):
79          if np.size(arrayA) == 1:
80              return self.close(np.array([arrayA, 0]), np.array([arrayB, 0]))
81          distance = np.sqrt(np.sum((arrayA[start:end] - arrayB[start:end])**2))
82          if distance > self.maxDistance:
83              return False
84          return True
85
86      def validPayment(self, payment):
87          for i in range(self.n):
88              totalCash       = self.assetVector[i]
89              for j in range(self.n):
90                  totalCash   += payment[j] * self.relativeLiabilityMatrix[j, i]
91
92              if (totalCash < payment[i] and not self.close(totalCash,
                ↪  payment[i])) or payment[i] > self.totalPaymentVector[i]:
93                  print("Invalid payment from node " + str(i))
94                  print("Total cash: " + str(totalCash))
95                  print("Clearing payment: " + str(payment[i]))
96                  print("Total payment: " + str(self.totalPaymentVector[i]))
97                  return False
98
99          return True
100
101     def countBankruptcy(self, payment):
102         bankrupt = 0
103         for i in range(self.n):
104             totalCash       = self.assetVector[i]
```

```
105             for j in range(self.n):
106                 totalCash   += payment[j] * self.relativeLiabilityMatrix[j, i]
107 
108             if totalCash < self.totalPaymentVector[i] and not
                ↪  self.close(totalCash, self.totalPaymentVector[i]):
109                 bankrupt += 1
110         return bankrupt
111 
112     def bestPayment(self):
113         paymentVector = np.array(self.clearingPaymentVector) +
            ↪  (2*self.maxDistance / np.sqrt(self.n))
114         return not self.validPayment(paymentVector)
115 
```

A.2 fictitiousDefaultAlgorithm.py

```
1  import numpy as np
2  from financialSystem import financialSystem
3  
4  class fictitiousDefaultAlgorithm():
5      def __init__(self, financialSystem):
6          self.relativeLiabilityMatrix    =
           ↪  np.array(financialSystem.relativeLiabilityMatrix)
7          self.n                          = financialSystem.n
8          self.totalPaymentVector         =
           ↪  np.array(financialSystem.totalPaymentVector)
9          self.assetVector                = np.array(financialSystem.assetVector)
10         self.defaultMatrix              = np.zeros((self.n, self.n))
11         self.clearingPaymentVector      = np.zeros(self.n)
12         self.financialSystem            = financialSystem
13 
14     def solve(self):
15         self.clearingPaymentVector      = self.totalPaymentVector
16         same = False
17         while not same:
18             self.updateDefaultMatrix()
19             newPaymentVector            = self.updatePaymentVector()
20             same                        =
               ↪  self.financialSystem.close(newPaymentVector,
               ↪  self.clearingPaymentVector)
21             if same:
22                 break
23             self.clearingPaymentVector  = newPaymentVector
24 
```

```
25          return self.clearingPaymentVector
26
27      def updatePaymentVector(self):
28          newPaymentVector =  np.dot(self.defaultMatrix,
            ↪  np.dot(np.transpose(self.relativeLiabilityMatrix), \
29                                (np.dot(np.identity(self.n) - self.defaultMatrix,
                                  ↪  self.totalPaymentVector) + \
30                                np.dot(self.defaultMatrix,
                                  ↪  self.clearingPaymentVector))) +
                                  ↪  self.assetVector) + \
31                                np.dot(np.identity(self.n) - self.defaultMatrix,
                                  ↪  self.totalPaymentVector)
32
33          return newPaymentVector
34
35      def updateDefaultMatrix(self):
36          totalCash    = self.assetVector + np.dot(self.clearingPaymentVector,
            ↪  self.relativeLiabilityMatrix)
37          bankrupt     = np.where(totalCash < self.totalPaymentVector)
38          solvent      = np.where(totalCash >= self.totalPaymentVector)
39
40          self.defaultMatrix[bankrupt, bankrupt]  = 1
41          self.defaultMatrix[solvent, solvent]    = 0
42
```

A.3 LPSolver.py

```
1   import numpy as np
2   from scipy.optimize import linprog
3   from financialSystem import financialSystem
4
5   class LPSolver():
6       def __init__(self, financialSystem):
7           self.financialSystem        = financialSystem
8           self.clearingPaymentVector  = np.zeros(self.financialSystem.n)
9
10      def solve(self):
11          c                           = np.ones(self.financialSystem.n) * -1
12          A_ub                        = np.eye(self.financialSystem.n) -
            ↪  np.transpose(self.financialSystem.relativeLiabilityMatrix)
13          b_ub                        = self.financialSystem.assetVector
14          bounds                      = [(0,
            ↪  self.financialSystem.totalPaymentVector[i]) for i in
            ↪  range(self.financialSystem.n)]
```

```
        self.clearingPaymentVector  = linprog(c=c, A_ub=A_ub, b_ub=b_ub,
        ↪  bounds=bounds).x

        return self.clearingPaymentVector
```

A.4 DQY_Algorithm.py

```
import numpy as np
from financialSystem import financialSystem
import math

class DQY_Algorithm():
    def __init__(self, financialSystem):
        self.relativeLiabilityMatrix    =
        ↪  financialSystem.relativeLiabilityMatrix
        self.n                          = financialSystem.n
        self.totalPaymentVector         = financialSystem.totalPaymentVector
        self.assetVector                = financialSystem.assetVector
        self.clearingPaymentVector      =
        ↪  np.array(financialSystem.totalPaymentVector)
        self.financialSystem            = financialSystem
        self.minPaymentVector           = np.zeros(self.n)
        self.maxPaymentVector           =
        ↪  np.array(financialSystem.totalPaymentVector)

    def solve(self):
        newPaymentVector = self.updatePaymentVector()
        while not self.financialSystem.close(newPaymentVector,
        ↪  self.clearingPaymentVector):
            self.helper(0)
            newPaymentVector = self.updatePaymentVector()
            if self.financialSystem.close(newPaymentVector,
            ↪  self.clearingPaymentVector):
                return self.clearingPaymentVector
            elif self.A_Above_B(newPaymentVector, self.clearingPaymentVector):
                self.minPaymentVector = np.array(newPaymentVector)
            elif self.A_Below_B(newPaymentVector, self.clearingPaymentVector):
                self.maxPaymentVector = np.array(newPaymentVector)
            for i in range(self.n):
                if self.maxPaymentVector[i] < self.minPaymentVector[i]:
                    temp = self.maxPaymentVector[i]
                    self.maxPaymentVector[i] = self.minPaymentVector[i]
                    self.minPaymentVector[i] = temp
```

```
32          self.clearingPaymentVector = newPaymentVector
33          return self.clearingPaymentVector
34
35      def helper(self, depth):
36          while True:
37              self.clearingPaymentVector[depth] = (self.minPaymentVector[depth] +
                ↪  self.maxPaymentVector[depth]) / 2
38              newPaymentVector = self.updatePaymentVector()
39              if (newPaymentVector[depth] - self.clearingPaymentVector[depth])**2
                ↪  <= (self.financialSystem.maxDistance**2)/self.financialSystem.n:
40                  if depth == self.n - 1:
41                      return
42                  else:
43                      self.helper(depth + 1)
44                      return
45              elif self.clearingPaymentVector[depth] < newPaymentVector[depth]:
46                  self.minPaymentVector[depth] = newPaymentVector[depth]
47              elif self.clearingPaymentVector[depth] > newPaymentVector[depth]:
48                  self.maxPaymentVector[depth] = newPaymentVector[depth]
49              if self.maxPaymentVector[depth] < self.minPaymentVector[depth]:
50                  temp = self.maxPaymentVector[depth]
51                  self.maxPaymentVector[depth] = self.minPaymentVector[depth]
52                  self.minPaymentVector[depth] = temp
53
54      def A_Below_B(self, arrayA, arrayB):
55          count = 0
56          for i in range(self.n):
57              if arrayA[i] < arrayB[i]:
58                  count += 1
59          return count == self.n
60
61      def A_Above_B(self, arrayA, arrayB):
62          count = 0
63          for i in range(self.n):
64              if arrayA[i] > arrayB[i]:
65                  count += 1
66          return count == self.n
67
68      def updatePaymentVector(self):
69          newPaymentVector = np.array(self.assetVector)
70          newPaymentVector += np.dot(np.transpose(self.relativeLiabilityMatrix),
            ↪  self.clearingPaymentVector)
71          newPaymentVector = np.minimum(newPaymentVector, self.totalPaymentVector)
72          return newPaymentVector
```

A.5 financialSystemWithShares.py

```
import numpy as np
from scipy.optimize import linprog
from financialSystem import financialSystem

class financialSystemWithShares(financialSystem):
    def __init__(self, n, k, liability, asset, costFunction, maxDistance=1,
    ↪ debtEdge=0, sharesEdge=0, quantityEdge=0, bCost=[],
    ↪ financialSystem=None):

        if financialSystem == None:
            super().__init__(n, liability, asset, maxDistance, debtEdge)
            self.k                      = k
            self.quantity               = np.zeros((self.n, self.k))
            self.quantityEdge           = quantityEdge
            self.asset                  = asset
            self.clearingPaymentMatrix  = np.zeros((self.n, self.n))
            self.maxDistance            = maxDistance
        else:
            self.n                      = financialSystem.n
            self.assetVector            = financialSystem.assetVector
            self.edge                   = financialSystem.edge
            self.liability              = financialSystem.liability
            self.liabilityMatrix        = financialSystem.liabilityMatrix
            self.relativeLiabilityMatrix =
            ↪ financialSystem.relativeLiabilityMatrix
            self.totalPaymentVector     = financialSystem.totalPaymentVector
            self.k                      = financialSystem.n
            self.quantity               = np.eye(self.n)
            self.quantityEdge           = 1
            self.asset                  = financialSystem.asset
            self.clearingPaymentMatrix  = np.zeros((self.n, self.n))
            self.maxDistance            = financialSystem.maxDistance

        self.shares             = np.zeros((self.n, self.n))
        self.sharesEdge         = sharesEdge
        self.solventMatrix      = np.eye(self.n)
        self.bankruptcyCost     = np.zeros(self.n)
        self.costFunction       = costFunction
        self.valueVector        = np.zeros(self.n)
        if np.size(bCost) == 3:
            self.bankruptcyCostB = bCost[0]
            self.bankruptcyCostC = bCost[1]
            self.bankruptcyCostD = bCost[2]

```

```
42
43      def generate(self):
44          if self.k < 1:
45              print("k must be greater than 0")
46              return
47          super().generate()
48          self.generateQuantity()
49          self.generateShares()
50
51
52      def generateQuantity(self):
53          self.quantityEdge               = max(0, min(self.quantityEdge, self.k))
54          self.assetVector                = np.random.normal(loc=self.asset,
            ↪  scale=np.sqrt(self.k), size=self.k).astype(np.float64)
55
56          self.quantity       = np.zeros((self.n, self.k))
57          mu                  = self.quantityEdge
58          sigma               = np.sqrt(self.k)
59          seed                = np.random.normal(mu, sigma, size=self.n)
60          seed                = np.maximum(np.zeros(self.n),
            ↪  np.minimum(np.round(seed), np.ones(self.n) * self.k))
61
62          for i in range(self.n):
63              position                    = np.random.choice(self.k,
                ↪  replace=False, size=int(seed[i]))
64              self.quantity[i, position]  = np.random.rand(int(seed[i]))
65
66
67      def generateShares(self):
68          self.sharesEdge             = max(0, min(self.sharesEdge, self.n - 1))
69
70          self.shares         = np.zeros((self.n, self.n))
71          mu                  = self.sharesEdge
72          sigma               = np.sqrt(self.n)
73          seed                = np.random.normal(mu, sigma, size=self.n)
74          seed                = np.maximum(np.zeros(self.n),
            ↪  np.minimum(np.round(seed), np.ones(self.n) * self.n - 1))
75
76          for i in range(self.n):
77              numberOfShares                      = int(seed[i]) + 1
78              position                            = np.random.choice(self.n - 1,
                ↪  replace=False, size=numberOfShares - 1)
79              position[np.where(position >= i)]   += 1
80              sharesVector                        = np.random.rand(numberOfShares)
81              sharesVector                        /= np.sum(sharesVector)
82              self.shares[i, position]            = sharesVector[1:]
```

```
83
84
85      def solve(self):
86          self.clearingPaymentMatrix  = np.array(self.liabilityMatrix)
87          self.valueVector            = np.zeros(self.n)
88          same                        = False
89          count =0
90          while not same:
91              count += 1
92              totalCash = self.calculateTotalCash()
93              self.updateSolventMatrix(totalCash)
94              self.updateBankruptcyCost(totalCash)
95
96              defaultMatrix                                       =
                ↪  np.where(self.solventMatrix.diagonal() == 0)
97              self.clearingPaymentMatrix[defaultMatrix, :]    = np.maximum(0, \
98
                                                                    ↪  self.relativeLiabilityMatrix[
                                                                    ↪  :] * \
99
                                                                    ↪  (totalCash[defaultMatrix,
                                                                    ↪  np.newaxis] -
                                                                    ↪  \
100
                                                                    ↪  self.bankruptcyCost[defaultMa
                                                                    ↪  np.newaxis]))
101             oldValue                                        =
                ↪  np.array(self.valueVector)
102             self.valueVector                                =
                ↪  np.dot(np.linalg.inv(np.eye(self.n) - np.dot(self.shares,
                ↪  self.solventMatrix)), \
103
                                                                    ↪  np.dot(self.quantity,
                                                                    ↪  self.assetVector)
                                                                    ↪  + \
104
                                                                    ↪  np.sum(self.clearingPaymentMa
                                                                    ↪  axis=0) - \
105
                                                                    ↪  self.totalPaymentVector
                                                                    ↪  -
                                                                    ↪  self.bankruptcyCost)
106             same                                            =
                ↪  super().close(oldValue, self.valueVector)
107         self.clearingPaymentVector = np.sum(self.clearingPaymentMatrix, axis=1)
108         return self.clearingPaymentVector, self.clearingPaymentMatrix,
            ↪  self.valueVector, count
```

```
109
110
111     def updateSolventMatrix(self, totalCash):
112         bankrupt                                = np.where(totalCash <
            ↪ self.totalPaymentVector)
113         solvent                                 = np.where(totalCash >=
            ↪ self.totalPaymentVector)
114         self.solventMatrix[bankrupt, bankrupt]  = 0
115         self.solventMatrix[solvent, solvent]    = 1
116
117
118     def updateBankruptcyCost(self, totalCash):
119         if self.costFunction == 'linear':
120             self.bankruptcyCost = self.bankruptcyCostC + self.bankruptcyCostD *
                ↪ totalCash
121         elif self.costFunction == 'logarithmic':
122             self.bankruptcyCost = np.exp(self.bankruptcyCostC) * (totalCash **
                ↪ self.bankruptcyCostD)
123             self.bankruptcyCost = np.maximum(np.ones(self.n) * 100,
                ↪ self.bankruptcyCost)
124         elif self.costFunction == 'quadratic':
125             if self.bankruptcyCostD != 0:
126                 totalCash = np.minimum(self.bankruptcyCostC / (-2 *
                    ↪ self.bankruptcyCostD), totalCash)
127             self.bankruptcyCost = self.bankruptcyCostB + self.bankruptcyCostC *
                ↪ totalCash + self.bankruptcyCostD * totalCash**2
128         elif self.costFunction == 'constant':
129             self.bankruptcyCost = np.ones(self.n) * self.valueVector
130         else:
131             return
132         self.bankruptcyCost = np.dot(np.eye(self.n) - self.solventMatrix,
            ↪ self.bankruptcyCost)
133
134
135     def calculateTotalCash(self):
136         return  np.dot(self.quantity, self.assetVector) + \
137                 np.sum(self.clearingPaymentMatrix, axis=0) + \
138                 np.dot(np.dot(self.shares, self.solventMatrix),
                    ↪ np.maximum(np.zeros(self.n), self.valueVector))
139
140
141     def validPayment(self, payment, output=False):
142         for i in range(self.n):
143             totalCash       = np.dot(self.quantity, self.assetVector)[i]
144             for j in range(self.n):
145                 totalCash   += payment[j] * \
```

```
146                             self.relativeLiabilityMatrix[j, i] + \
147                             (self.solventMatrix[j, j] * self.shares[i, j] *
                                ↪ np.maximum(0, self.valueVector[j]))
148
149             if (totalCash < payment[i] and not super().close(totalCash,
                ↪ payment[i])) or payment[i] > self.totalPaymentVector[i]:
150                 if output:
151                     print("Invalid payment from node " + str(i))
152                     print("Total cash: " + str(totalCash))
153                     print("Clearing payment: " + str(payment[i]))
154                     print("Total payment: " + str(self.totalPaymentVector[i]))
155                 return False
156
157         return True
158
159     def bestPayment(self):
160         paymentVector = np.array(self.clearingPaymentVector) +
            ↪ (2*self.maxDistance / np.sqrt(self.n))
161         return not self.validPayment(paymentVector)
162
163     def countBankruptcy(self, payment):
164         bankrupt = 0
165         totalCash = self.calculateTotalCash()
166         for i in range(self.n):
167             if totalCash[i] < self.totalPaymentVector[i] and not
                ↪ self.close(totalCash[i], self.totalPaymentVector[i]):
168                 bankrupt += 1
169         return bankrupt
170
```