Algorithms for Computing Clearing Payments in Financial Networks

Alvaro Martin-Angulo Martin

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

We study algorithms to compute *clearing payments* in *financial networks* interconnected by financial contracts in different settings: When there are only direct debt between financial institutions; When bankruptcy costs are also present; and finally with the inclusion of Credit Default Swap contracts.

Our implementations of the algorithms for the different settings provide interesting insights into how efficient some of the algorithms are in a *meaningful* context, where we try to reproduce somewhat realistic financial networks based on research.

We find the Linear Programming Algorithm we implement has the best runtime performance in the *meaningful* networks we generate, despite not having a polynomial worst-case runtime, due to the nature of these networks being *hyper-sparse*. Optimisations for the Linear Programming Algorithm for hyper-sparse networks make it more efficient than the bounded Fictitious Default Algorithm for example.

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.

(*Alvaro Martin-Angulo Martin*)

Acknowledgements

Firstly, I would like to thank Prof. Kousha Etessami for his continued support and guidance throughout this challenging project. I am particularly grateful for the freedom and trust he has instilled in me. He has allowed me to take the project in the direction I found most interesting, while providing valuable insights in every step of the way.

Next, I would like to highlight my friends and flatmates, who have made this challenging year one to remember. Looking forward to being back home and having a laugh with them is something invaluable to me.

My family and my parents have been incredibly supportive throughout not only this project, but in anything that I have set to do in life. For that I cannot thank them enough, and it makes me an incredibly proud child to continue to live up to all the trust and love they have given me.

Lastly, I want to acknowledge *Tapia De Casariego*, and my friends from this beautiful place. The feeling of being in *Tapia* is something indescribable, and spending time there fuels me to power through any challenges I can face.

Contents

Chapter 1

Introduction

> *Every firm owed money to every other firm. But ... you couldn't tell whether they were bankrupt or not, because that depended on whether they got paid money that was owed to them by other firms who might or might not be in default, depending on whether the firms that owed them money went bankrupt [28].*

A *Financial Network* is a way of modelling financial institutions that are interconnected by financial contracts between them. When one or more of the institutions in the network defaults[1], some of the institutions may no longer be able to pay their obligations to the other institutions. In turn, this may have a ripple effect in the network, and the institutions that do not receive their expected payment fully may also not be able to fulfill their debt[2] obligations, and hence also default.

The quote above by Nobel laureate Joseph Stiglitz describes an example situation where this occurred. During the East Asia crisis (1997-1998), where 70% of firms in Indonesia went into default, more than 50% in Korea, and almost 50% in Thailand, it was hard to establish the value of any firm - as they depended on whether they got payed money that was owed to them by other firms.

This problem motivates the question tackled in this thesis: when there is financial distress, and some institutions in a financial network may default, what has to be payed between institutions to *clear* the financial network?

Our task is to find the *clearing payments* for a given financial network, which represent the total payment each institution has to make to *clear* the network. We consider *payments* to be *clearing* if they are in accordance to standard conditions imposed by bankruptcy law, that is - they satisfy the conditions of proportional repayments of liabilities[3] in default, limited liability of equity[4], and absolute priority of debt over equity [7].

[1] Default: to fail to do something, such as pay a debt, that you legally have to do.

[2] Debt: something, especially money, that is owed to someone else, or the state of owing something

[3] Liability: the fact that someone is legally responsible for something.

[4] Equity: The money value of a property or business after debts have been subtracted.

As in [7], throughout this thesis we assume that all payments are made at the same time. In practice, this is far from the truth, as it is virtually impossible to get institutions to pay exactly at the same time, and values of contracts and debts vary constantly. Clearing payments for such cases, namely dynamic financial networks, have also been studied by [2], [5].

Since the landscape of research in this field is vast, with financial networks represented with many different types of contracts between them, and in different contexts, we have to restrict the settings of the networks we consider in this thesis. We constructively change the setting of the financial networks considered, making the task of finding *clearing payments* harder. For each of the settings considered, we *implement* and *experiment* with *algorithms* to compute said payments.

First, in **Chapter 2** we provide the formal definitions of the settings of the networks considered in this thesis. For each of the settings, we provide background information on the research done on algorithms to compute clearing payments for said settings.

Chapter 3 then considers the most simple setting, as studied by [7], where financial networks *only* have debt contracts between them. We implement three algorithms for computing clearing payments for this setting, namely: the *Iterative Solver*, the *Fictitious Default Solver*, and the *Linear Solver*. We then introduce a small variation to the setting, with the addition of *bankruptcy costs* into the network, as done by [24]. This slightly changes the task of computing the clearing payments, as when institutions default, they lose part of their value to account for the costs associated with bankruptcy. For this setting, we implement the *Greatest Clearing Vector Solver*, provided by [24], which we can compare to the other algorithms in this chapter by setting no bankruptcy costs. We provide theoretical worst time complexities for our implementations, and perform experiments on the performances of algorithms, with both randomly generated networks, and meaningful networks.

In both of the settings above, the *existence* of a *clearing payment* is *guaranteed*, and we provide algorithms to compute the clearing payments in polynomial time. With the addition of Credit Default Swap[5] (CDS) contracts into the network, these guarantees no longer hold. [25] show that computing an approximate solution to the *clearing* problem with sufficient small constant error is *PPAD-complete*. This reveals that computational complexity can become a concern regarding the stability of financial networks. This is the setting we study in **Chapter 4**, where we add CDS contracts into the network of debt and bankruptcy costs. We consider the restrictions to the network provided in [26], which allow for computing clearing payments for this setting in polynomial time, implement algorithms to compute said payments, and perform experiments with them.

Lastly, in **Chapter 5** we provide our conclusions to the thesis, in regards to the practical relevance of our study, the limitations, and the possible extensions.

[5]CDS are a form of financial derivative that depend on three institutions: a creditor, a debtor and a reference institution. The debtor agrees to pay the creditor a certain amount if the reference institution defaults.

Chapter 2

Background and Definitions

In this section, we provide the formal definitions of the different types of financial networks we study in this thesis. We restrict this thesis to three types of networks: networks with debt contracts only; networks with debts contracts and default costs; and networks with debt contracts, default costs, and credit default swaps. We summarise and evaluate the available papers with respect to finding algorithms for computing clearing payments in these kind of networks, to then provide implementations for said algorithms in chapter [3] and [4].

2.1 Financial Networks with Debt Contracts Only

In their paper, which can be considered as a base in the literature of this topic, upon which further investigation has stemmed in many directions, [7] introduce a network of financial institutions which are interconnected *only* by debt obligations between them. They prove that in this setting, clearing payments *always* exist, that under mild regulatory conditions, the clearing payment vector is unique, and they provide an algorithm to compute the clearing payment vector, namely the *Fictitious Default Algorithm*.

2.1.1 Framework

We now introduce the framework provided by [7], adjusting notation where suitable to fit the thesis, allowing for the extensions we provide in the later sections.

The financial network is built of a set of N financial institutions. Each of these institutions $i \in N$ may have liabilities towards other institutions within the network, which are represented by an N x N nominal liabilities matrix L. The nominal liabilities matrix L is defined as a non-negative N x N matrix for which L_{ij} represents the nominal liability of node i to node j. They define an external cash flow vector e which represents the cash infusion to the node from sources outside the financial system. The external cash flow vector e is an N-dimensional vector for which $e_i \geq 0$ represents the cash flow for node i.

They let p be the *total payment vector*, for which each p_i represents the total payment

of node i to other nodes in the network. They define the *total obligations vector* $\overline{p}$ as an N dimensional vector for which $\overline{p}_i$ represents the *total obligations* of node i to all other nodes:

$$\overline{p}_i = \sum_{j=1}^{n} L_{ij}$$

They let Π be the *relative liabilities matrix*, which for each node represents the fraction of the node's total obligations it has towards another node.

$$\Pi_{ij} = \begin{cases} \frac{L_{ij}}{\overline{p}_i} & \text{if } \overline{p}_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Under this framework, the *value of equity of node i* is given by the total cash flow minus the payments to creditors:

$$\sum_{j=1}^{n} \Pi_{ij}^{T} p_j + e_i - p_i$$

Definition 1. Debt Only Financial Network

(a) A *Debt Only Financial Network* is defined as a tuple (*N, L, e*) where N is a set of banks, L is a nominal liabilities matrix, and e is an external cash flow vector.

(b) Equivalently, a *Debt Only Financial Network* is defined as a tuple (N, Π, $\overline{p}$, e) where N is a set of banks, Π is a *relative liabilities matrix*, $\overline{p}$ is a *total obligations vector* and e is an external cash flow vector.

2.1.2 Clearing Payment Vectors

Intuitively, a clearing payment vector should represent the payments made by each of the nodes in the financial system. In their paper, [7] consider that these payments must be consistent with legal rules allocating value among nodes and holders of debt and equity. These are:

(1) **Limited liability**: Total payments of a node must not exceed the cash flow available to the node.

(2) **Priority of debt claims**: Stockholders in the node receive no value until node is able to pay off all of it's liabilities.

(3) Proportionality: If default occurs, claimant nodes are paid by the defaulting node in proportion to the size of their nominal claim on firm assets.

Under this desiderata, a clearing payment vector p^*, has to be consistent with the following properties:

(1) Limited Liability: $\forall i \in \{1, ..., N\}$,

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

(2) Absolute Priority $\forall i \in N$ either obligations are paid in full ($p_i^* = \overline{p}_i$), or all value is paid to creditors:

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

Definition 2. Debt Only Clearing Payment Vector

A *Debt Only Clearing Payment Vector* for a *Debt Only Financial Network* (N, Π, $\overline{p}$, e) is a vector $p^* \in [\mathbf{0}, \overline{p}]$ that is a fixed point of the function ϕ, that is $\phi(p^*) = p^*$. Where $\phi : [0, \overline{p}]^N \to [0, \overline{p}]^N$ is defined by:

$$\phi_i(p) \equiv \begin{cases} \overline{p}_i & \text{if } \overline{p}_i \leq \sum_{j=1}^{n} \Pi_{ij}^T p_j + e_i \\ e_i + \sum_{j=1}^{n} \Pi_{ij}^T p_j & \text{otherwise.} \end{cases}$$

From the above, we see that a clearing payment vector is one where every node pays the minimum of what it has and what it owes.

From a fixed point argument, since the function ϕ is linear, and monotone on the complete lattice, it has a lattice of fixed points, a greatest and a least fixed point, implying that a clearing payment vector always exists [16]. Next, [7] prove that under mild regularity conditions for the network, the clearing payment vector is unique (we omit this condition, as the proof is lengthy, and in practice, networks almost always meet the condition, refer to [7] for the explanation). We are interested in finding *maximum clearing payment vectors*, As these maximise the value of equity of all the nodes in the network.

They provide an algorithm, namely the *Fictitious Default Algorithm* to compute the clearing payment vector in financial networks where there exists a unique clearing payment vector. This algorithm produces a sequence of vectors starting at $p^0 = \overline{p}$, and reduces to the clearing payment vector p^* in at most N iterations. We provide an implementation of such algorithm and compare it to other algorithms that also solve this problem in section (3.1).

2.2 Financial Networks with Debt Contracts and Bankruptcy Costs

As an extension to [7], [24] provide a financial network where they include a bankruptcy cost as a fraction of the equity available to the financial institution at the time of default. This arguably sets a more realistic scenario, as financial institutions have to incur bankruptcy costs when they file bankruptcy. These costs include direct costs of paying lawyers and accountants, and indirect costs for fire sales or lost profits [31]. [21] study the default costs as actual functions and not just constant factors, arguably making the setting more realistic as it gives room for more precise interpretations of default costs. In this thesis, we focus on [24], as it is then extensible to the Credit Default Swap setting in section (2.3).

They prove the existence of clearing payment vectors in such networks, and provide a variation of the *Fictitious Default Algorithm* to not only find the clearing payment vector under the mild regularity conditions, but for *any* financial network of this kind.

2.2.1 Framework

Building from the financial system of (2.1), [24] introduce two constants, $\alpha, \beta \in (0,1]$ where α is the the fraction of the face value of net assets realized on liquidation, and β is the fraction of the face value of inter-bank assets realized on liquidation. This means, upon bankruptcy, a financial institution will pay the fraction α of their external cash flow, and the fraction β of the payments they receive from other banks as the clearing payment. The outstanding amount is assumed to be the cost associated to bankruptcy of that institution. That is, the bankruptcy cost for a node i in default is:

$$(1-\beta)\sum_{j=1}^{n}\Pi_{ij}^{T}p_j^{*}+(1-\alpha)e_i.$$

Definition 3. Debt and Bankruptcy Costs Financial Network

A *Debt and Bankruptcy Costs Financial Network* is defined as a tuple $(N,\Pi,\overline{p},e,\alpha,\beta)$ where N is the set of banks, Π is the Relative Liabilities Matrix, $\overline{p}$ is the total obligations vector,e is the external cash flow vector, and $\alpha,\beta \in (0,1]$ the fraction of their external assets, and the fraction of the payments they receive from other banks associated with bankruptcy costs upon defaulting, respectively.

2.2.2 Clearing Payment Vectors

Under the same desiderata as those in [2.1] (limited liability, priority of debt claims, and proportionality), and with the introduction of bankruptcy costs, the clearing payment vector can be defined as follows:

Definition 4. Debt and Bankruptcy Costs Clearing Payment Vector

A *Debt and Bankruptcy Costs Clearing Payment Vector* for a *Debt and Bankruptcy Costs Financial Network* $(N,\Pi,\overline{p},e,\alpha,\beta)$ is a vector $p^* \in [\mathbf{0},\overline{p}]$ such that $p^* = \phi(p^*)$ (a fixed point), where ϕ is the function defined by:

$$\phi_i(p) \equiv \begin{cases} \overline{p}_i & \text{if } \overline{p}_i \leq \sum_{j=1}^{n}\Pi_{ij}^{T}p_j + e_i \\ \alpha e_i + \beta\sum_{j=1}^{n}\Pi_{ij}^{T}p_j & \text{otherwise.} \end{cases}$$

In this setting, they show that the function ϕ is monotone and increasing, as well as bounded above by $\overline{p}$. This is enough to show that the clearing payment vector always exists, and is a fixed point of the ϕ function.

In their papers, both [24] and [21] extend their research to *bailout costs*. By looking at which nodes default at certain points in the network, they provide insights into these types of payments, which essentially consist in *"rescuing"* banks in the network by acquiring their debt. This itself is another topic within the clearing payment problem,

which we don't go into in this thesis, but leave as a possible extension if we had more time.

Interestingly, [24] provide an algorithm to find the *maximum* clearing payment vector in at most N iterations in *any* financial network of this kind (in contrast to [7] who's algorithm works under mild regulatory conditions). We implement this algorithm, namely the *Greatest Clearing Vector Algorithm* in section (3.2.2).

2.3 Financial Networks with Credit Default Swaps

With the introduction of *Credit Default Swaps* (CDS) in a financial network, the problem of finding a clearing payment vector becomes non-trivial. In the two network cases looked at above, (2.1, 2.2), algorithms are devised to compute clearing payments efficiently (in polynomial runtime). [25] prove that when introducing CDS into the network, finding an approximate solution for a clearing payment vectors is PPAD-Complete. This implies that the problem does not have a polynomial time approximation scheme (PTAS) unless P=PPAD and thus needs to be considered computationally intractable. [20] build on this paper to actually prove that finding a strongly approximate solution is FIXP-Complete[1]. Furthermore, [26] show that in the unbounded CDS case, with bankruptcy costs, there can also be no solution to the clearing problem, or the solution can be *ambiguous*, meaning that there are multiple conflicting solutions, none of which maximise the equity of the network, making it impossible to pick a solution that will make all the financial institutions "happy".

Building on [24], [26] introduce the concept of credit default swaps into the financial network, and devise sufficient restrictions for the existence of solutions to the clearing payment problem in this setting, as well as providing algorithms to find them. By constructively applying less restrictions, they provide solutions to different scenarios where it is possible to find a clearing payment vector, trying to approximate to the unbounded case as much as possible. This is done through the *Colored Dependency Graph* framework, which models a directed graph of *long* and *short* positions of the CDS network.

2.3.1 Framework

The CDS framework devised by [26] builds on [7] and [24], adding the CDS contracts into the network. To make it more understandable, we first explain the concept of a CDS and then show how these contracts are modeled in the network.

2.3.1.1 Credit Default Swaps

CDS contracts involve three parties: a debtor, a creditor, and a reference entity. The creditor and debtor agree on a *notional* value for the contract. Upon default of the

[1] Input: algebraic circuit (straight-line program) over basis +, , , /, max, min with rational constants, having n input variables and n outputs, such that the circuit represents a continuous function $F : [0,1]^n \implies [0,1]^n$ (The domain can be much more general than $[0,1]^n$) Output: Compute a (ε-near approximate) fixed point of F. [11]

reference entity, the debtor has to pay the creditor depending on the *recovery rate* of the reference entity. That is, the proportion of the liabilities it is able to pay back. The amount the debtor pays to the creditor is the *notional* of the contract times the proportion of the liabilities the reference entity is not able to pay.

To model the CDS contracts in the network, fist recall Definition 3, for a *Debt and Bankruptcy Costs Network*. This is a tuple $(N, \Pi, \overline{p}, e, \alpha, \beta)$, which can be represented equivalently by the tuple (N, L, e, α, β). We now introduce a three dimensional matrix c, the *CDS matrix*, which is an N x N x N matrix for which $c^k_{i,j}$ represents the value of the CDS contract where i is the debtor, j is the creditor, and k the reference entity.

Definition 5. CDS Network A *CDS Network* is defined as a tuple $(N, L, c, e, \alpha, \beta)$ where N is the set of banks, L is the Nominal Liabilities Matrix (debt contracts), c is the CDS matrix, e is the external cash flow vector, and $\alpha, \beta \in (0, 1]$ the constants associated with bankruptcy costs.

Now, it is important to define the *recovery rate vector* r, for which $r_i = 1$ if bank i is not in default and $r_i < 1$ if bank i is in default.

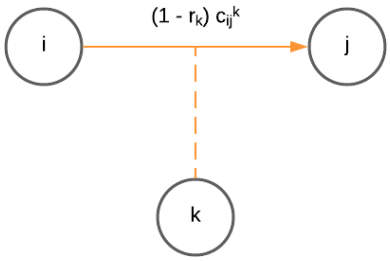


Figure 2.1: Example graph of a CDS contract, where i is the debtor, j is the creditor, and k is the reference entity. Upon default of k, i will pay j: $(1 - r_k) \cdot c^k_{ij}$ for this contract.

The *liabilities* of bank i to bank j under *recovery rate vector* r can be defined as the direct debt plus the value of the credit default swaps of i to j:

$$l_{ij}(r) = L_{ij} + \sum_{k=1}^{N} (1 - r_k) \cdot c^k_{i,j}.$$

Which follows that the *total liabilities* of a bank i are the sum of the liabilities to all other banks:

$$l_i(r) = \sum_{j=1}^{N} l_{ij}.$$

However, the actual payment $p_{ij}(r)$ a bank makes can be lower than the liability $l_{ij}(r)$ if bank i is in default under r. Due to this, following the proportionality condition discussed in 2.1.2, the payments have to be proportional to the total liability of the bank:

$$p_{ij}(r) = r_i \cdot l_{ij}.$$

The total assets a_i of a bank under a *recovery rate vector r* consist of the external cash flow and the incoming payments, which are:

$$a_i(r) = e_i + \sum_{j=1}^{N} p_{ji}(r).$$

If bank i is in default, their *assets after default costs* $a_i'(r)$ are:

$$a_i'(r) = \alpha \cdot e_i + \beta \cdot \sum_{j=1}^{N} p_{ji}(r).$$

2.3.2 Clearing Recovery Rate Vector

Following the above, we are able to define a clearing recovery rate vector. We take the legal rules from (2.1.2) as conditions, and devise the following:

Definition 6. CDS Clearing Recovery Rate Vector

A *CDS Clearing Recovery Rate Vector* for a *CDS Network*$(N, L, c, e, \alpha, \beta)$ is a vector $r^* \in [\mathbf{0}, 1]$ such that $r^* = \phi(r^*)$ (a fixed point) where $\phi : [0,1]^N \to [0,1]^N$ is defined by:

$$\phi_i(r) = \begin{cases} 1 & \text{if } a_i \geq l_i \\ \frac{a_i'(r)}{l_i(r)} & \text{otherwise} \end{cases}$$

In essence, the recovery rate vector is 1 if i is not in default, and the assets after the default costs are incurred over the total liabilities otherwise. From this definition, we can obtain a clearing payment vector p^*, for which the entries of the vector would be: $p_i^* = \sum_{j=1}^{N} p_{ij}(r^*)$

In this setting, the liabilities depend on the *recovery rate vector r*, and the assets have terms of the form $c_{ij}^k \cdot r_j \cdot (1 - r_k)$, meaning that the update function $\phi_i(r)$ depends on r in a way that is *nonlinear* and *non-monotonic*: an increase in some recovery rate r_h could lead to an higher or lower $\phi_i(r)$ for another bank i [26]. These properties make computing a fixed point in this setting hard, and impossible in some cases.

Chapter 3

Algorithms for Debt Only, and Debt and Bankruptcy Cost Networks

In this chapter, we look at how to compute clearing payments for financial networks with debt contracts only, and financial networks with debt contracts and bankruptcy costs. We group these two types of networks into one chapter due the ability to represent both networks with the same kind of *liabilities matrix*, which makes testing and comparing algorithms possible. This becomes apparent in the explanation of section (3.4).

For each of the types of networks, we explain the available algorithms to compute clearing payments, provide our implementation details, and finally a runtime analysis of each of these. We then look at comparing the effectiveness of each of the algorithms in differently generated networks: completely random, and meaningful networks.

3.1 Debt Only

We first look at algorithms for computing clearing payments in financial networks with debt contracts only. All the algorithms implemented in this section come from the paper on Systemic Risk in Financial Systems provided by [7]. They provide a description of the *Fictitious Default Algorithm*, which we implement, and provide a *programming characterization* of the problem of finding a clearing payment vector, for which we implement a *Linear Programming Solver*. Finally, we implement an algorithm that simply iterates the function ϕ until it converges to a solution.

3.1.1 Linear Programming Solver

We now look at one of the ways of computing clearing payments in *Debt Only Financial Networks*: Linear Programming. This is a mathematical modelling technique used to maximize or minimize a linear function subject to various constraints [3].

In their paper, [7] provide a linear programming characterisation for *Debt Only Financial Networks*. This is based on maximising the payments by all nodes in the system subject to the limited liability condition. They prove that any function $f : [0, \overline{p}] \mapsto \mathbb{R}$ that is

strictly increasing under the linear programming characterization will produce a clearing payment vector. The linear programming problem then has the following form:

$$\text{Max } f(p) \text{ s.t. } p \leq \Pi^T p + e \text{ and } 0 \leq p \leq \overline{p}$$

To solve this using Python, we use Scipy's *linprog* function from the optimize library [30]. They provide interfaces to the linear programming solvers HiGHS simplex and HiGHS interior-point. The function picks between the two, depending on the problem. HiGHS is a linear programming software, and the methods employed come from [18].

The LINPROG function only minimizes an objective function, so we have to adapt our problem to fit it. The function optimizes problems of the form:

$$\text{Min } c^T x \text{ s.t. } Ax \leq b \text{ and } l \leq x \leq u$$

To adapt our problem to this format, we set c to be a vector with values -1. This essentially means we minimise the negative values of x, effectively maximising them. Rearranging the equation then gives us

$$\text{Min } -p \text{ s.t. } I_N - \Pi^T p \leq e \text{ and } 0 \leq p \leq \overline{p}$$

The output p is the clearing payment vector.

3.1.2 Iterative Algorithm

Recall the definition of a *Clearing Payment Vector* p^* for a *Debt Only Financial Network*, a fixed point of the function ϕ, defined by:

$$\phi(p_i) \equiv \begin{cases} \overline{p}_i & \text{if } \overline{p}_i \leq \sum_{j=1}^n \Pi_{ij}^T p_j + e_i \\ e_i + \sum_{j=1}^n \Pi_{ij}^T p_j & \text{otherwise.} \end{cases}$$

One way to compute the clearing payment vector is to start by setting $p = \overline{p}$, and simply iterating ϕ. This would produce a sequence of vectors where $p^{(n+1)} = \phi(p^{(n)})$. We can observe two important properties in ϕ:

(1) ϕ is bounded above by $\overline{p}$: for any p, $\phi(p) \leq \overline{p}$;

(2) ϕ is monotone: if $\hat{p} \leq p$, then $\phi(\hat{p}) \leq \phi(p)$.

[7] provide a proof of the above. With these two properties, since $p^{(1)} \leq p^{(0)} = \overline{p}$, for all n:

$$p^{(n+1)} < p^{(n)}.$$

Since $p^{(n)}$ are all non-negative, it follows that there is a monotone limit

$$p' :=\downarrow \lim_{n\to\infty} p^{(n)}.$$

Also, notice that the set $A_n := \{i : p_i^{(n)} < \overline{p_i}\}$ is non decreasing in n, and eventually constant. The set A_n is a set for each iteration n consisting of the indices of the vector

$p^{(n)}$ for which the value $p_i^{(n)} \leq \overline{p_i}$. The set is non-decreasing because $p^{(n+1)} \leq p^{(n)}$, meaning that the indices in the set can only increase or stay constant. Hence, $p^{'}$ satisfies $p^{'} = \phi(p^{'})$ (a fixed point), showing that $p^{'}$ is a clearing payment vector. This proof is inspired on that provided in [24].

Our implementation for this algorithm is straight forward: start at $p = \overline{p}$, and work out *newP* by the following: $newP = \phi(p)$. The algorithm runs in a while loop until $p = newP$, at which point we have come to a fixed point.

Algorithm 1 Iterative Solver

function ITERATIVESOLVER(*financialNetwork*)
    $p \leftarrow financialNetwork.p\_bar$
    $newP \leftarrow$ GETNEWP$(p, financialNetwork)$
    while $p \neq newP$ **do**
        $p \leftarrow newP$
        $newP \leftarrow$ GETNEWP$(p, financialNetwork)$
    end while
end function

3.1.3 Fictitious Default Algorithm

Following the Linear Programming solver, we implement the algorithm explicitly provided by [7]: *The Fictitious Default Algorithm*.

The idea behind this algorithm is to sequentially identify the banks that default, by assuming that at the start of each round, the banks that haven't yet defaulted are not in default and pay fully. When the algorithm gets to a round where no new banks default, the algorithm terminates and the clearing payment vector is obtained. Refer to algorithm (2)

Algorithm 2 Fictitious Default Solver

function FICITITIOUSDEFAULT(*financialNetwork, p, defaultMatrix*)
    $newDefaultMatrix \leftarrow$ GETDEFAULTMATRIX$(p, financialNetwork)$
    $newP \leftarrow$ GETNEWP$(newDefaultMatrix, p, financialNetwork)$
    if $oldDefaultMatrix = newDefaultMatrix$ **then**
        return p
    else
        return FICITITIOUSDEFAULT(*financialNetwork, newP, newDefaultMatrix*)
    end if
end function

For this algorithm, [7] introduce the concept of the *default matrix* $\Delta(p)$. This is defined as a diagonal N-dimensional matrix for which the diagonal entries are 1 if the node is in default, and 0 otherwise.

$$\Delta(p)_{ij} = \begin{cases} 1 & i = j \text{ and } i \text{ is in default} \\ 0 & \text{otherwise} \end{cases}$$

We implement a function GETDEFAULTMATRIX which loops through all the nodes, and for each node calculates if it is in default by working out if their cash flow and incoming payments are greater than $\overline{p}_i$. To find the new payment vector under the new default matrix, we implement a function GETNEWP which sets p_i to $\overline{p}$ if the node is not in default, and to the total value of the node otherwise. The algebra for this is taken from [7]:

$$newP = \Delta(p)(\Pi^T(\Delta(p)p + (I - \Delta(p))\overline{p}) + e) + (I - \Delta(p))(\overline{p}).$$

To run the algorithm, we start with a default matrix of all 0s, and set $p = \overline{p}$. The algorithm will recursively identify the defaulting nodes and arrive to the clearing payment vector in at most N iterations.

3.2 Bankruptcy Costs

After looking at the algorithms implemented for *Debt Only Financial Networks*, we now introduce the default costs from [24]. Recall the definition of a *Debt and Bankruptcy Costs Financial Network* as a tuple $(N, \Pi, \overline{p}, e, \alpha, \beta)$.

3.2.1 Discontinuity does not allow for linear programming solver

Recall from the previous section that when introducing bankruptcy costs, the total payment bank will be $\overline{p}$ if the bank has enough money available, and will pay a discounted amount from their total value if it does not have enough.

$$p_i = \begin{cases} \overline{p}_i & \text{if } \overline{p}_i \leq \sum_{j=1}^n \Pi_{ij}^T p_j + e_i \\ \alpha e_i + \beta \sum_{j=1}^n \Pi_{ij}^T p_j & \text{otherwise} \end{cases}$$

If we look at what happens at the exact point when $p_i = \overline{p_i}$, we see that a discontinuity occurs. if we decrease p_i by 1 cent, p_i will jump from being $\sum_{j=1}^n \Pi_{ij}^T p_j + e_i$ to $\alpha e_i + \beta \sum_{j=1}^n \Pi_{ij}^T p_j$, in other words, there is a jump of $(1-\beta)\sum_{j=1}^n \Pi_{ij}^T p_j + (1-\alpha)e_i$. This discontinuity makes the use of linear programming impossible: we can no longer have the constraint that $p \leq \Pi^T p + e$ because in that interval, there will be a discontinuity if a bank defaults, nor can we claim that $p \leq \beta \Pi^T p + \alpha e$, because this avoids banks that can pay in full to do so.

3.2.2 Greatest Clearing Vector Algorithm

As a way of solving the issue caused by this discontinuity, which does not allow for the linear programming characterization, as compared to the *Debt Only Financial Network* case, [24] provide the *Greatest Clearing Vector Algorithm*. This algorithm builds from the idea of the *Fictitious Default Algorithm* discussed in section 3.1.3. In essence, the idea behind it is the same, where in each iteration, you calculate the set of defaulting banks, until this set does not change, coming to a solution in at most N iterations.

However, the discontinuity in the function does not allow for us to calculate *newP* in such a straight forward manner. At each iteration, we calculate a default set, and fix the payments of the banks not in default in that iteration to $\overline{p}_i$. For the banks in default, we solve a system of linear equations where we consider the bankruptcy factors α and β.

We now provide the steps for the algorithm, adapting those provided by [24] to suit our notation.

(1) Set $p = \overline{p}$, *defaultSet* = $\emptyset$

(2) Under p, calculate for all nodes $i \in N$ the nodes that are in default, that is, i is in default if

$$\sum_{j=1}^{N} p_j \cdot \Pi_{ji} + e_i - \overline{p}_i < 0$$

If this is the case, add index i to *newDefaultSet* (the default set of this iteration)

(3) Define the *nonDefaultSet* as a set of the indices of the banks that are not in default, i.e. $nonDefaultSet := \{i \quad | \quad i \in \{1,2,\ldots,N\} \cap i \notin defaultSet\}$

(4) If the default set doesn't change (*newDefaultSet* = *defaultSet*), terminate the algorithm and return p.

(5) Otherwise, set the payments of banks that are not in default to $\overline{p}$

$$newP_i = \overline{p}_i \quad \forall i \in nonDefaultSet$$

And determine the remaining values for *newP* by solving the system of linear equations that takes into consideration the default costs, as these banks are in default. Let nonD = nonDefaultSet, and newD = newDefaultSet (For presentation purposes below)

$$newP_j = \alpha e_j + \beta[\sum_{j \in nonD} \overline{p}_j \Pi_{ji} + \sum_{j \in newD} newP_j \Pi_{ji}] \quad \forall j \in newD$$

(6) set $p \leftarrow newP$ and *defaultSet* $\leftarrow$ *newDefaultSet* and go back to step 2.

To implement this in Python, most of the steps are straight forward, however we need to adapt the format of the equations to be able to solve the system of linear equations of step 5. We now provide the overview of our recursive implementation, a brief discussion on implementation decisions, and then explain how we solve the system of linear equations.

For this implementation, we use python sets to represent the *defaultSet*, *newDefaultSet* and *nonDefaultSet*. Since at each iteration we need to loop through the defaulting and non defaulting nodes, we use sets to iterate through them, rather than lists, because we can access exactly the items that we need, which will take *O(Number of items in set)*. In a list, we would need to iterate through the whole list every time, meaning *O(N)* every time we want to loop through one of the sets. In the worst case, where one set contains all items, it will also take *O(N)*, and that is the overlaying time complexity, but in the general case it will perform slightly better.

Algorithm 3 Greatest Clearing Vector Solver

```
function GREATESTCLEARING(network, p, defaultSet)
    newDefaultSet ← GETDEFAULTSET(p, network)
    if defaultSet == newDefaultSet then
        return p
    end if
    nonDefaultSet ← all i ∉ newDefaultSet
    newP ← p
    for i in nonDefaultSet do
        newP[i] ← network.p̄[i]
    end for
    newP ← LINEARSOLVE(nonDefaultSet, defaultSet, network)
    return GREATESTCLEARING(network, newP, newDefaultSet)
end function
```

3.2.2.1 Solving the Set of Linear Equations

To solve the set of linear equations in algorithm 3, we implement a function LINEARSOLVE, which leverages numpy's LINALG.SOLVE function to solve the system of equations.

3.3 Theoretical Worst Case Time Complexities of Algorithms

In this section, we provide the theoretical worst case time complexities of the algorithms implemented in this chapter. Their practical performance is then evaluated in the following section.

We use N: the number of banks in the input as the variable to perform our analyses. Interestingly, we find that the theoretical time complexities of the *fictitious default algorithm* and the *greatest clearing vector algorithm* are equal, when we expected that the *greatest clearing vector algorithm* may have had a greater time complexity due to solving a system of linear equations at each iteration.

This makes the practical analysis in 3.4 interesting, as we see how algorithms with the same theoretical time complexity behave in different settings.

3.3.1 Linear Programming Solver

The overlaying time complexity of this algorithm is determined by the SCIPY.OPTIMIZE.LINPROG function provided by SciPy[30]. The work done before, as explained is section (**??** is purely rearranging the equation into a form that is understood by said function. The operation with the most significant time complexity in the rearranging is the subtraction of Π from the identity matrix. Since Π is of size N x N, the operation of matrix subtraction is in the order of $O(N^2)$.

Determining a worst case time complexity for linear programming algorithms is complex, and is a field that is evolving and being studied at present. For the sake of not diverting the content of this thesis, we will not perform a deep study of this or how they work, and rather provide brief ideas of what the complexities could look like. We base our ideas from the textbook on Understanding and Using Linear Programming [22], which you can refer to for substantial information on this topic.

The methods SCIPY.OPTIMIZE.LINPROG use are simplex and interior-point. The simplex method has no known polynomial upper bound to the worst-case runtime at the date of writing this thesis. The best known bound for this method is $O(2^n)$ where n is the number of the variables for the linear program. In our case that would be $O(2^N)$. Some of the interior-point methods are known to be (weakly) polynomial, with a time complexity bounded by $O(n^3L)$ where n is the number of variables in the input and L is the maximum bit size of coefficients in the linear program [22]. This would be $O(N^3L)$ in our case.

However, [22] explain that seeing worst-case behaviours in linear programming methods rarely occurs, so seeing instances as bad as those described above is expected to be rare. Rather, the usual runtimes are bounded by constants or $O(log(n))$. We therefore focus on looking at how often these behaviours occur, as well as how well these methods perform through an empirical analysis in section 3.4.

3.3.2 Iterative Algorithm

Recall from 3.1.2 that this algorithm has an indefinite number of iterations to get to the clearing payment vector. Our proof shows that the clearing payment vector is achieved as n (the iteration) tends to infinity: $\lim_{n\to\infty} p^{(n)}$.

From the above, we cannot devise an asymptotic worst time for the algorithm, as the steps it will take to the clearing payment vector are indefinite. Our analysis can just look at the time complexity of finding a new p at each iteration, which is $O(N^2)$ since we loop through the matrix Π of size N x N.

An empirical analysis is more interesting here, to see the behaviour of this algorithm in practice, and the usual number of steps it takes to come to a clearing payment vector. This is done in section (3.4).

3.3.3 Fictitious Default Algorithm

Recall from (3.1.3), that the algorithm comes to a clearing payment vector in at most N iterations. To find the time complexity, we look at our implementation in algorithm 2.

We see that this algorithm works in a recursive manner, and we know that in the worst case, the algorithm will be called *O(N)* times. Now we look at the work that is done inside of each of the algorithm calls.

Fist, we call GETDEFAULTMATRIX. The time complexity of the function is $O(N^2)$, determined by looping through the complete 2d-matrix Π of size N x N. for each of the

elements we loop through, the operations done are of constant time, and hence we can ignore them (integer addition, multiplication, inequality checks).

Then, we call GETNEWP. This function in essence performs a large matrix vector multiplication. Since we are multiplying vectors such as p of size N by matrices like Π, of size N x N, the overlaying time complexity of this function is of $O(N^3)$

The final operation done is to check for matrix equality with two N x N matrices. This is simply done in $O(N^2)$ time.

Finally, the overall time complexity of the algorithm is $O(N) \cdot O(N^2 + N^3 + N^2) = O(N) \cdot O(N^3) = O(N^4)$.

3.3.4 Greatest Clearing Vector Algorithm

We follow a similar procedure as above. Recall that this algorithm is also recursive, and arrives at the clearing payment vector in at most *O(N)* calls. We now look at the operations done at each of the calls to find the complexity of the algorithm. Refer to algorithm 3 for this explanation.

First, we call GETDEFAULTSET, which loops through the matrix Π, meaning once again $O(N^2)$ complexity for this operation.

We then compare the default set matrices of size N x N, resulting in $O(N^2)$ complexity.

Creating the *nonDefaultSet*, and *newP* are both done in $O(N)$ time each. Similarly, the following for loop also has a time complexity of $O(N)$ as it loops through the *nonDefaultSet*, of size of at most $N-1$ (Not all nodes can default in a system).

The last operation is calling LINEARSOLVE, to solve the system of linear equations remaining. To rearrange the values into a format for the function NP.LINALG.SOLVE, $O(N^2)$ time is required.

The NP.LINALG.SOLVE function itself, provided by NumPy, uses the LAPACK (Linear Algebra Package, a library for linear algebra) gesv routine to calculate the solutions[17]. As shown in the LAPACK Benchmarks, solving an *n-by-n* system of linear equations with 1 right hand side, has a time complexity of $O(n^3)$. In our case, the size of the input matrix will be of at most size N (if no nodes are in default), meaning that we can expect a worst case runtime of $O(N^3)$.

Overall, the time complexity of the algorithm is: $O(N) \cdot O(N^2 + N^2 + 3N + N^2 + N^3) = ON \cdot O(3N + 3N^2 + N^3) = O(N^4)$

3.4 Experiments: Network Generation and Effects on Performance

In this section, we perform an empirical study of the performance on the algorithms implemented. We first generate completely random networks, varying the number of banks in the networks and having liabilities and equities take values between 1 to 10000.

We then generate more meaningful networks, by looking at studies of what usual debt network structures look like, and see the effect this has on the different algorithms.

Keeping the machine constant to run these tests is fundamental, as the central processing unit (CPU)'s power has a direct impact on the runtime of algorithms. For this thesis, we use the laptop model name: ROG Zephyrus G15, CPU: AMD Ryzen 9 5900HS with Radeon Graphics 3.30 GHz, and RAM: 32.0GB.

3.4.1 Randomly Generated Networks

To generate random networks, we use NumPy's random module[17], which contains a function RANDINT that allows the generation of pseudorandom integers for a range of values, and the dimensions required. We create an *N* x *N liabilities matrix* of random integers with this function, as well as the *N* dimensional *cash flow vector*. Finally, we use NumPy's FILL_DIAGONAL function to fill the diagonal entries of the liabilities matrix with zeros (banks do not have liabilities towards themselves).

Since both *Debt Only Networks* and *Debt and Bankruptcy Cost Networks* have the same kind of liabilities matrix L, and cash flow vector e, we can generate these in common for both networks. Recall that *Debt and Bankruptcy Cost Networks* have two constants α and β as their factors of discount for bankruptcy. If we set both of these factors to 1: $\alpha = \beta = 1$, we effectively create a *Debt Only Network*, as there is no discount upon bankruptcy. This is interesting because it allows us to compare how the *Greatest Clearing Vector Algorithm* performs against the three other algorithms implemented for *Debt Only Networks* when there are no bankruptcy costs. We also look at how the *Greatest Clearing Vector Algorithm* performs with changing α and β for common liabilities matrix L and common e against algorithms for *Debt Only Networks*.

3.4.1.1 Rounding Errors

The first thing we wanted to look at was the *correctness* of our implementations. Even though in our theoretical discussion we always arrive at an exact clearing payment vector, in practice this does not occur. This is due to the representation of rational numbers in computers. Our algorithms use NumPy's DOUBLE data type. This is a 64-bit double-precision floating point number, preferred over FLOAT as this is only 32-bit, and not precise enough, and over LONGDOUBLE which is 128-bit, due to the time it would take for the algorithms to perform operations on the data[17].

We ran two experiments to see what affects the rounding errors the most. First, we fix the values that liabilities and cash flows can take to be an integer from 1 to 10000, and look at the effect of the network size on rounding errors. For the second experiment we fix the number of nodes to 100, and look at the effect of increasing the possible values cash flows and liabilities can take.

The rounding errors we found present themselves in the clearing payments not being exact, and leaving some of the nodes in the network with negative equity after the clearing payment is applied (this would imply that they would need to pay more money than they have. In theory this should not happen, and banks that default should have

an equity of 0 after the clearing payment, as they should pay out all their values to creditors.

To analyse the significance of these errors, we work out the *average percentage error*. To define the aforementioned, we need to recall the definition of the *value of equity of nodes* in *Debt Only Financial Networks* from 2.1.1 as

$$\sum_{j=1}^{n} \Pi_{ij}^{T} p_j + e_i - p_i$$

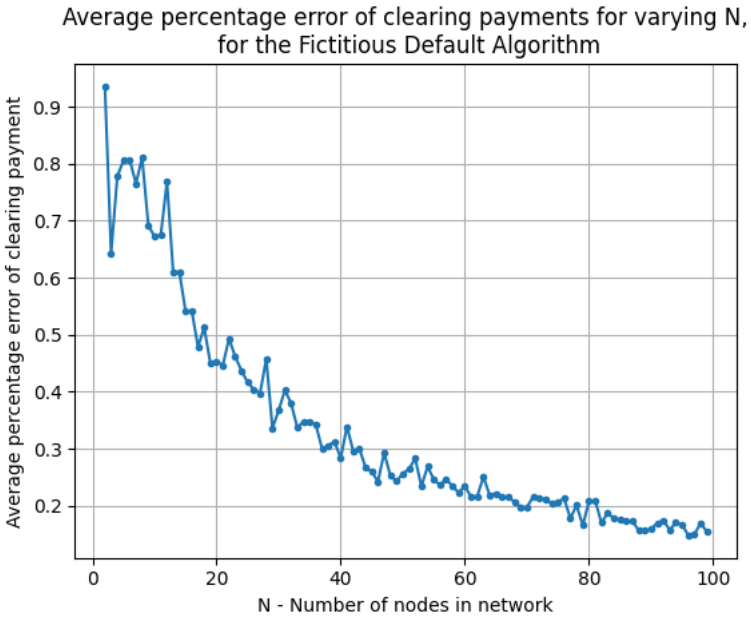
We call the *value of equity of nodes* after applying the clearing payment vector p^* produced by the algorithms the *equity after clearing* e^*. For each index i in p^*, the percentage error is 0 if $e_i^* >= 0$, and $\frac{|e_i^*|}{p_i^*}$ otherwise. The average of these values is the *average percentage error*. This value gives us a rough idea of how significant the errors in the clearing payments are.

Figure (3.1) shows that the fictitious default algorithm has significant rounding errors (almost 1%), particularly for networks of small size. Taking an average percentage error of 1% would imply, that on average, every node in the network would have to pay an extra 1% over the clearing payment vector found. This becomes a problem particularly for banks in default: if a bank defaults, it pays out all it's value to creditors, so how are they supposed to pay an extra 1% if all their value has been payed out? We may think that sharing this *outstanding* debt through the nodes in the network would perhaps be a way of tackling this problem, but then banks that are not in default would ask: why would I have to pay a percentage outstanding from a rounding error in an algorithm? The main problem is such a high magnitude of percentage error. When dealing with large amounts of capital, such as millions of dollars, a 1% is simply not justifiable for an algorithm error.

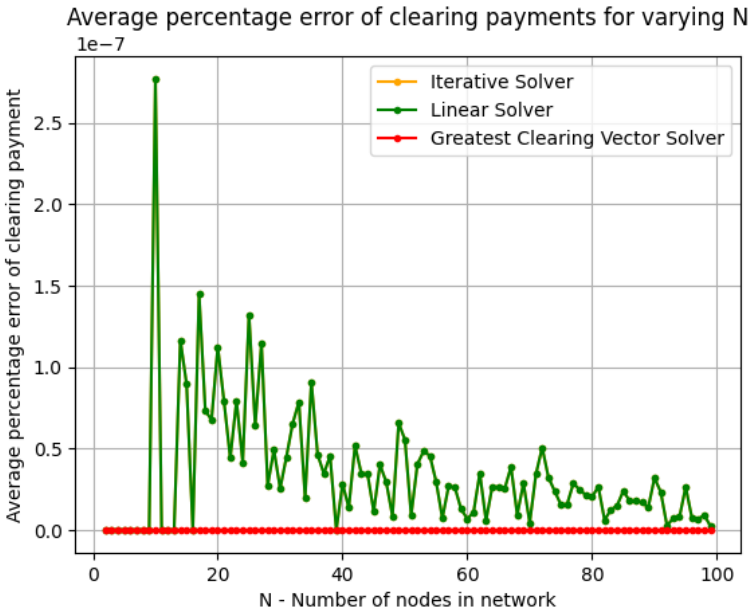
When looking at Figure (3.1) (b), we see that the linear solver has somewhat random, but very small percentage errors (in the magnitude of $10^{-}7$). This can perhaps be justified, since it is so small, and payed out by nodes in the network by sharing this outstanding amount. We see however, that the *Greatest Clearing Vector Solver*, and the *Iterative Solver* yield percentage errors of 0% for any $N \in \{2, ..., 100\}$. This makes them superior algorithms in the context of *correctness*. We suggest that further research is done, with more nodes in the network, to see if that affects how the algorithms behave for larger networks. We also suggest investigating other possible implementations of the vector matrix multiplications performed for the *Fictitious Default Solver*, which may lead to lower percentage errors, and make the algorithm practical.

3.4.1.2 Performances

To calculate the performances of our implementations, we use Python's *time* package [13]. The function TIME.TIME() returns the *epoch* (the epoch is the point where the time starts, it is January 1, 1970, 00:00:00 (UTC) on all platforms[13]) time in seconds as a float value. For each of the algorithms, we store the *startTime* as TIME.TIME() just before running the algorithm, and then store *endTime* as TIME.TIME() straight after running the algorithm. We then get the $runTime = endTime - startTime$, which gives



(a) Figure shows average percentage error over 100 iterations of the Fictitious Default Solver for N from 2 to 100.



(b) Figure shows average percentage error over 100 iterations of the three other algorithms for N from 2 to 100.

Figure 3.1: (a) and (b) are plotted separately due to the magnitudes of the percentage errors for the Fictitious Default Solver being more than 10^6 more significant than those for the algorithms in (b). In (a), we see that as the number of nodes increases, the percentage error of the Fictitious Default Algorithm decreases in what looks to be an exponential decay. In (b), we see that the Greatest Clearing Vector Solver, and the Iterative Solver yield 0% average percentage error for all N, while the Linear Solver has very small errors that seem to appear in a somewhat random manner.

us the execution time for the algorithm in seconds. By repeating iterations of algorithms, and finding average *runTimes*, we get the results for this subsection.

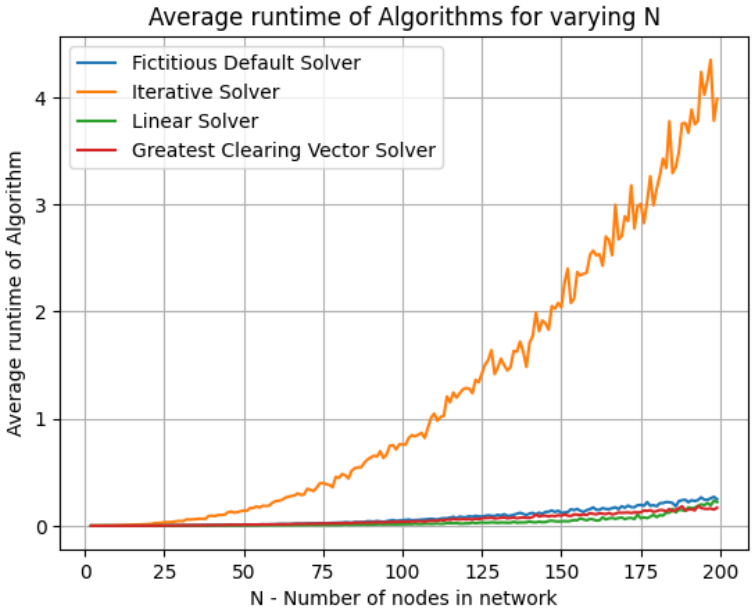
We first look at our results for the experiments where we have constant range of values for cash flows and range of values for liabilities, and we vary the number of nodes N in the network in figure (3.2). We see that due to the unbounded nature of both the *Linear Solver* and the *Iterative Solver*, their performance is significantly worse than the *Fictitious Default Solver*, and the *Greatest Clearing Vector Solver*. What is perhaps more surprising, is that in figure (3.2) (b), we see that the *Greatest Clearing Vector Algorithm* has an edge on the Fictitious Default Solver. This is surprising because even though the *Greatest Clearing Vector Algorithm* was designed to solve *Debt and Bankruptcy Cost Networks*, and also provides extra guarantees that it will always compute the maximum clearing payment vector for *any* network of this kind, as opposed to the *Fictitious Default Algorithm*, which needs *regularity* conditions, the *Greatest Clearing Vector Algorithm* has a better runtime.

We hypothesize this is due to how the *Fictitious Default Algorithm* calculates *newP* in each iteration (refer back to Algorithm (2)). This calculation requires four matrix vector multiplications, for matrices of size N x N and vectors of size N. This implies a worst case time complexity of $O(4N^3)$, which simplifies to $O(N^3)$, but in practice is slower than solving the system of linear equations required in every step for the *Greatest Clearing Vector Solver*. Experimenting with the time function to see if this is actually true, and diving deeper into each algorithm is required to confirm this relation, which we leave as a possible extension as future work. We point to research on *Optimizing the Performance of Sparse Matrix-Vector Multiplication* by [19], whose implementations are 3.1 times faster for a single vector and 6.2 times faster for multiple vectors. As we see in the following section, *meaningful networks* are very sparse, and can benefit from said optimisations.

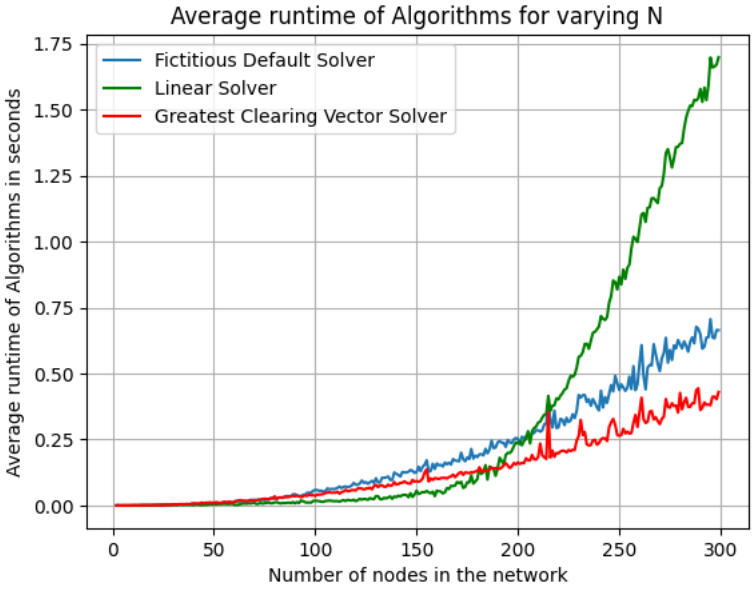
3.4.2 Generating Meaningful Networks

Generally speaking, getting information on debts institutions have towards others is a hard task, as this type of information is usually private. The most *meaningful* data we could find about real world debt contracts between institutions is that provided in a publicly available study of global market for inter-bank *syndicated loans*[4]. Syndicated loans are loans offered by a group of loaners[27]. This study takes the data for syndicated loans from 1980-2007 provided in [14]. Given the composition of the syndicate, they create bank-to-bank links (simulating direct debt relationships) between participating banks. To construct the network, each bank-to-bank link in a deal is replicated as many times as there are lenders in the syndicate, and equal loan amounts are assigned to each participating lender in the deal[4].

The finding we are interested in for the time frame between 1980-2007 is the *average degree* of nodes (3.34), and the number of nodes in the network (4806). The average degree of nodes here refers to the average number of edges nodes have in the graph (average in-degree and average out-degree have to be the same). In the previous subsection (3.4.1), our *liability matrices* are very *dense* (edges between most of the nodes), since the entries are modeled as a random number, meaning very little entries



(a) Figure shows average runtimes over 100 iterations of the four algorithms (Fictitious Default Solver, Iterative Solver, Linear Solver, Greatest Clearing Vector Solver) for N from 2 to 200.



(b) Figure shows average runtimes over 100 iterations of the three algorithms (Fictitious Default Solver, Linear Solver, Greatest Clearing Vector Solver) for N from 2 to 300

Figure 3.2: (a) shows that the Iterative Solver becomes highly inefficient in comparison to the other three algorithms as N increases, due to the unbounded nature of the algorithm (No bounded number of iterations to come to a solution). In (b), we focus on the other three algorithms, and see that the Linear Solver has a similar behaviour as the Iterative Solver in (a). We also see that the Greatest Clearing Vector Solver performs slightly better than the Fictitious Default Solver, even though their theoretical worst case time complexities are the same $O(N^4)$.

will actually be 0. With the found data, we provide a new approach to creating more *meaningful* liability matrices.

To create *liability matrices* of dimensions N x N, we create a function GENERATE_MEANINGFUL(N). This function generates liability matrices with nodes having an average degree based on [4]. The findings from the study suggest that the usual *average degree* of nodes is $\simeq 3.4$, despite there being less nodes in the network (From 1995-2007, 2861 nodes, Avg.degree of 3.43; From 2001–2007, 1486 nodes, Avg. Degree 3.72), the average degree does not seem to decrease. To mimic this behaviour, we define a function $f(i,j)$ that defines the value of an edge from $i \to j$. GENERATE_MEANINGFUL(N) loops through each $(i,j) \in N^2$ pair where $i \neq j$, and sets the value of $L_{ij} = f(i,j)$, where $f(i,j)$ is the function defined by:

$$f(i,j) = \begin{cases} 0 & \text{with probability } (1 - \frac{3.4}{N}) \\ V & \text{with probability } \frac{3.4}{N} \end{cases}$$

Where V is a randomly generated number such that $V \in \{1,\ldots,M\}$, and M is the maximum possible value of an edge for the network.

To implement $f(i,j)$, we first generate a random number with NP.RANDOM.RANDOM(), which generates a float between 0 and 1. We check if this random number is less than $\frac{3.4}{N}$, and if it is, we set the value of the edge $L_{ij} = V$, and to 0 otherwise. This mimics the idea of the probability of there being an edge being $\frac{3.4}{N}$. The last step is to check that every node has at least one outgoing or incoming edge, since if it does not, it would not be a part of the network. We are generating networks where existence of edges is not guaranteed due to the nature of the generation being random, and therefore this check is vital. To do so, we loop through each $i \in N$, and for each $j \in N$, we check if there is an incoming or outgoing edge between (i,j). We add an edge from i to a random edge $k \in N$, where $i \neq k$, with a random value V when the following holds:

$$\sum_{j=1}^{N} L_{ij} + L_{ji} = 0.$$

For networks with $N < 3$, the above function does not make sense, as it will generate fully connected networks every time, leading to the random case we had in the previous subsection. We also do not study networks of less than 100 nodes in our experiments for this subsection, because extrapolating the value of *average degree* of 3.4 from [4] would be unrealistic, as the minimum number of nodes in their study is 1486.

3.4.2.1 Performances

When testing the four algorithms for the runtimes in meaningful networks, we find that some networks make the *iterative Solver* have extremely slow runtimes of over 3000 seconds for less than 200 nodes, refer to figure (3.3). We therefore plot the other three algorithms, in figure (3.4).

The results obtained are very interesting, as we see the *Linear Programming Solver* outperforms both the *Fictitious Default Solver* and the *Greatest Clearing Vector Solver*,

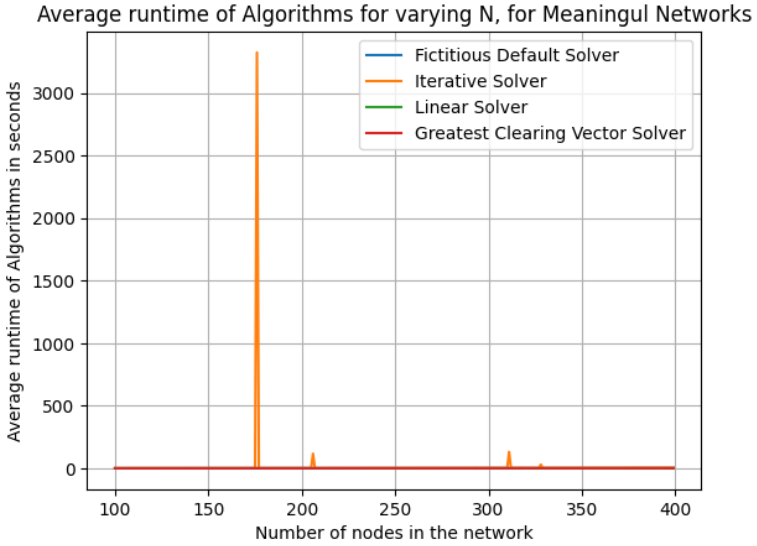


Figure 3.3: Meaningful Network showing a *Iterative Solver* can find instances that take unreasonable amounts of time (over 3000s for less than 200 nodes)

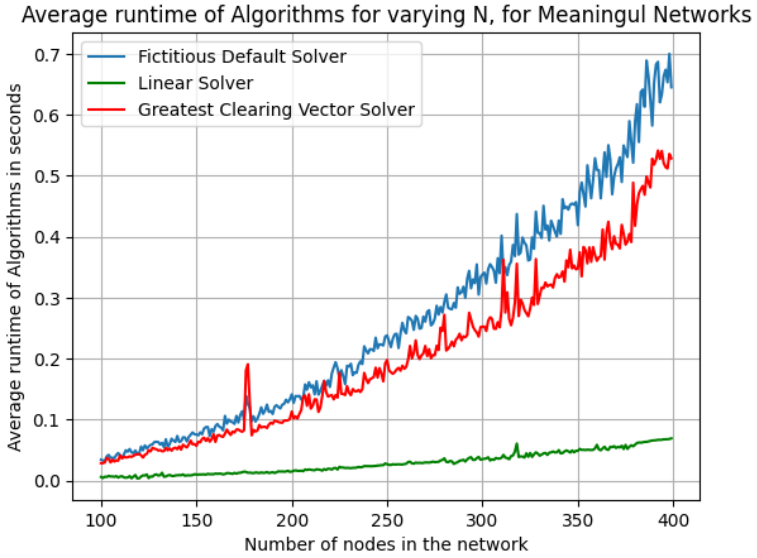


Figure 3.4: The figure shows how the Linear Solver has significantly better average runtimes over the 100 iterations for each *N*, than both the Fictitious Default Solver and the Linear Solver for Meaningful Networks of size $N \in \{100, ..., 400\}$

as compared to the previous section, where we saw the *Linear Solver* had much worse performance for $N > 200$ in figure (3.2).

The results obtained are justified by the nature of the networks we are testing. Our so called *meaningful networks* are very *sparse* (contain many *0* entries), as we make them have an average degree of around 3.4. When we compare this to the *randomly generated networks* in section (3.4.1), we see that *meaningful networks* are much more sparse than *randomly generated networks*. Such good performance of the *Linear Solver* for these networks is due to the implementation of the Linear Programming algorithms

by HiGHS being optimized for large sparse networks [18].

We now provide a brief explanation based on [18], but suggest referring to the paper for more detail. When the density of the coefficient matrix (average number of non-zero entries per column) is less than 10%, we can consider a matrix being *hyper-sparse* [15]. The *liability matrices* we deal with in our *meaningful networks* have on average *3.4* non-zero entries per column, meaning that for *meaningful networks* where $N > 34$, we can consider the *liabilities matrix* to be *hyper-sparse*. [15] analyse the revised simplex method, and show how when the *hyper-sparsity* property is present, it is highly inefficient. They provide techniques to exploit this property, proving an average increase in speed of 5.61 times when *hyper-sparsity* is present. These techniques are used in the HiGHS linear programming solvers, showing why our implementation of the *Linear Solver* is so effective for *meaningful networks*.

We now look at how having default costs affects the runtime of the *Greatest Clearing Vector Solver*. For our results here we run the *Greatest Clearing Vector Solver* with no default costs ($\alpha = \beta = 1$), with random default costs ($\alpha, \beta \in [0, 1]$), and fixed default costs ($\alpha = \beta = 0.99$), for the same networks. The values of 0.99 are taken to represent somewhat "realistic" bankruptcy costs, as [31] show that bankruptcy costs are on average around 1% of the value of the firm prior to bankruptcy. We realise these factors of α and β do not fully encompass this, as they may not be exactly represent a bankruptcy cost of 1%, but we leave this as a possible extension for future work, where α and β can be made functions of the value of the bank.

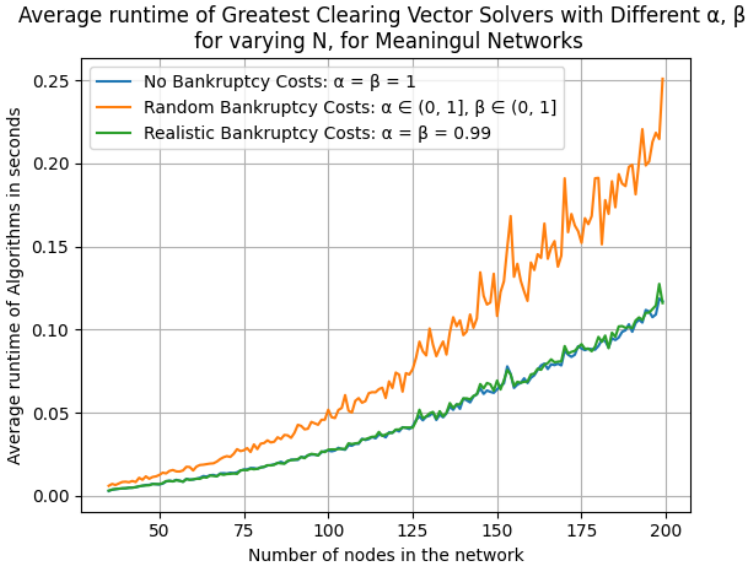


Figure 3.5: Figure shows how the average runtime over 100 iterations for each N of the *Greatest Clearing Vector Solver* with different bankruptcy costs performs for $N \in [35, ..., 200]$

As can be seen in figure (3.5), the averge runtime for the *Greatest Clearing Vector Solver* with no bankruptcy costs, and realistic bankruptcy costs performs almost the same, with little deviations. The random bankruptcy costs however have a worse performance overall. We hypothesize this might be because when there are higher bankruptcy costs

(smaller α and β), nodes that default pay out less value, which may cause a larger cascade of defaults of other nodes compared to when the bankruptcy cost is smaller. This relationship is confirmed in figure (3.6), where we run the *Greatest Clearing Vector Solver* with $\alpha = \beta \in [0.1, 0.2, ..., 1]$, and see how the runtimes for lower α and β generally show slower performances, proving our hypothesis above.

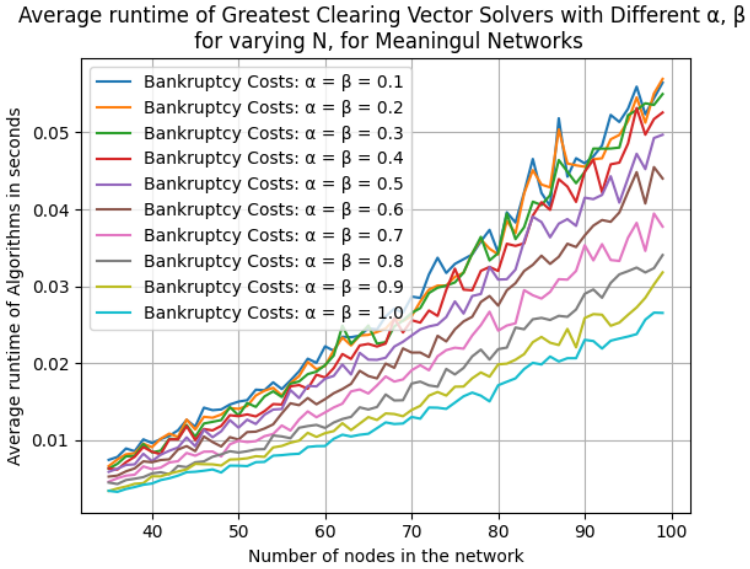


Figure 3.6: Figure shows how the average runtime over 100 iterations for each N of the *Greatest Clearing Vector Solver* with different bankruptcy costs performs for $N \in [35, ..., 200]$. We see how the lower the bankruptcy cost factors, the worse the performance.

Chapter 4

Algorithms for Networks with Debt, Bankruptcy Costs and Credit Default Swaps

We now look at networks with CDS contracts in them. Recall definition (5) of a *CDS Network* as a tuple $(N, L, c, e, \alpha, \beta)$. Building from the previous chapter, the variation here is the addition of the CDS contracts through the three-dimensional *CDS matrix*. It is important to also recall that when working with this type of network, we look for a *recovery rate vector*, and can devise a *clearing payment vector* from it.

We implement and explain the *Colored Dependency Graph* framework devised by [26], a way of constructing a directed graph from a *CDS Network* of the different types of Credit Default Swaps: Covered and Naked.

The *Colored Dependency Graph* is used to find if certain restrictions are true for the network. [26] provide three restrictions to the *Colored Dependency Graph* that allow for computing clearing payment vectors for the respective *CDS Networks* efficiently: Acyclic Colored Dependency Graph; Green Core Colored Dependency Graph; No Red-Containing Cycle Colored Dependency Graph. Due to time constraints, we consider the first two restrictions in this thesis. We implement algorithms to find if these restrictions are met, and implement algorithms to compute the clearing payment vectors when the restrictions are met. We provide implementation and design decisions at every step, and finally analyse the worst case runtime complexities of our implementations.

If we had more time, we would provide experimentation with generation of *CDS Networks* and their effect on the performance of our implemented algorithms. We realise that to do a thorough experimentation, similar to that in the previous chapter in section (3.4), more time is needed. Generating *meaningful* CDS Networks is a much more complex task, because in each CDS contract, three parties are involved. Generating fully random networks here would not make sense, as our algorithm depend on certain restrictions to be met.

4.1 Colored Dependency Graph

In this section, we explain the *Colored Dependency Graph* framework from [26]. We first introduce the concept of *covered* and *naked* CDS, which are the core of building this graph. We then explain how the graph is built, and provide our implementation decisions and details. Following this, we implement algorithms to find if the *colored dependency graphs* meet any of the three conditions provided by [26] for the possibility to compute a clearing payment efficiently. We do not focus on the proofs that these conditions make the computation possible, as they are all provided by them in [26], and rather the practical implementations of finding if networks meet these conditions in the most effective way possible.

4.1.1 Covered and Naked CDS

To introduce the concept of covered and naked CDS, we first explain the terms *long* and *short* positions. Informally, an entity i is said to be *long* towards another entity j if i would not benefit from a bad financial situation of entity j. Conversely, an entity i is short towards another entity j if i would benefit from a bad financial situation of j.

A *covered* CDS is one in which the creditor is long towards the reference entity. This implies that the sum of the notional values of the CDS contracts with that creditor and that reference entity are less than the direct debt the reference entity has towards the creditor. The creditor is *long* on the reference entity because it would not benefit from a bad financial situation of the reference entity. If the reference entity defaults, the creditor will get payed the notional of the CDS contracts, which are less than or equal to the value of the direct debt the reference entity had towards it. Otherwise, the creditor is short on the reference entity, and the CDS contract is considered *naked* [26].

Definition 7. Covered and Naked CDS For a *CDS Network* $(N, L, c, e, \alpha, \beta)$, a bank j has a *covered CDS position* towards another bank k if

$$\sum_{i=1}^{N} c_{ij}^{k} \leq L_{kj}.$$

Otherwise, j has a *naked* position towards k, that is

$$\sum_{i=1}^{N} c_{ij}^{k} > L_{kj}.$$

In the following graphs, direct debt contracts are represented as blue arrows between nodes, and CDS contracts as orange arrows between nodes, from which a dotted orange line points to the reference entity for that contract.

4.1.2 Green and Red Edges from CDS Network

In essence, a *Colored Dependency Graph* is a directed graph of green and red edges, for which a green edge represents a long position, and a red edge represents a short

position. An important remark is that the arrow points in the opposite direction than one would expect. If j is long on i, then there is a green edge from i to j.

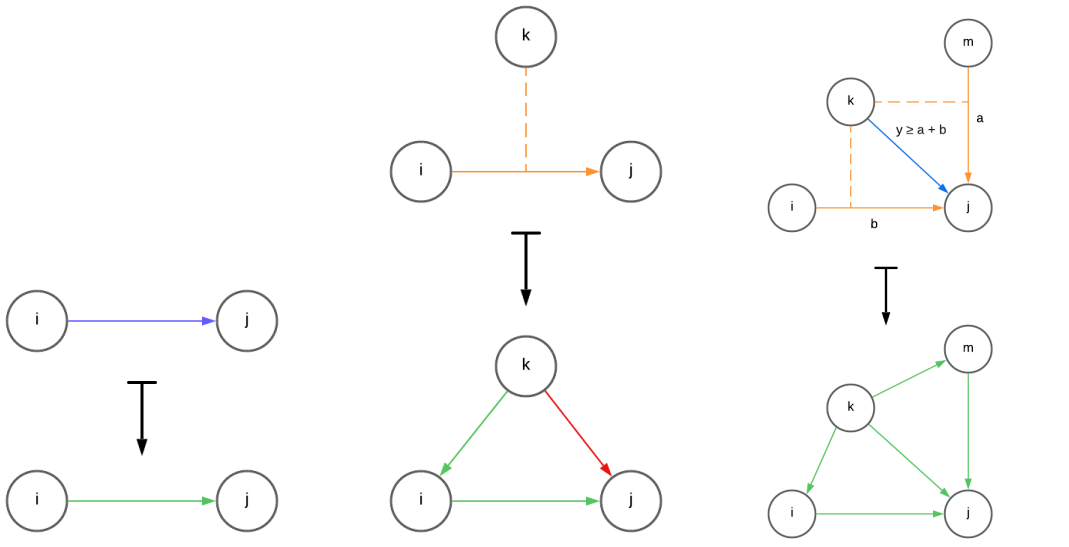
By looking at the types of contracts available (direct debt, covered CDS, and naked CDS) and how they translate into a *Colored Dependency Graph* makes this more apparent. We take the ideas from [25] to build figure 4.1, which helps understand how these graphs are built. We now proceed to explain each of the sub-figures.

(a) **Direct debt contract**: Intuitively, if i has direct debt towards j, then j is long on i. There is therefore a green edge from i to j

(b) **Naked CDS contract**: Similar to above, if j is the creditor for a CDS contract, and i the debtor, then j is long on i. i is long on k because if k is in a worse economic situation and defaults, i has to pay for the CDS contract with reference entity k. j is short on k because it would benefit from a bad economic situation of k, as this could activate the CDS contract where j is the creditor and k the reference entity.

(c) **Covered CDS contract**: In this sub-figure, we represent two nodes that have a CDS contract with creditor j and reference entity k to illustrate the importance of considering *all* CDS contracts with the same reference and creditor to determine if the creditor is long or short on the reference.

In this case, the creditor j is long on the reference entity k because the sum of the CDS contracts where j is the creditor and k the reference entity $(a+b)$ is less than or equal to the direct debt k has towards j ($y \geq a+b$). The other long positions between nodes follow from explanations provided above.



(a) Direct debt contract (b) Naked CDS Contract (c) Covered CDS Contracts

Figure 4.1: Figure showing how the different types of contract are translated into a *Colored Dependency Graph*.

4.1.3 Initialising a Colored Dependency Graph from a CDS Network

To implement Colored Dependency Graphs, we create a class `ColoredDepndencyGraph` which contains the red edges and green edges.

4.1.3.1 Representations of Green and Red Edges

We represent red edges and green edges as two separate dictionaries, where the keys are the indexes of the node where the edge is coming from, and the values are the set of nodes that have an incoming edge from that node. This could be of the form: *greenEdges = {0: set(1, 4, 5), 1: set(3, 2)}*. We use the defaultdict class from the collections library in Python, which allows for adding elements to a key which may not yet exist in the dictionary (creates it automatically), which avoids the work in checking for existence. We choose to implement the dictionaries this way to improve the runtimes of the algorithms we implement in the following sections. Since we are not interested in any kind of ordering, storing the nodes in this way rather than in a 2d array avoids us doing $O(N^2)$ work every time we want to find the edges of the graph (it will be $O(N^2)$ time complexity, but in the general case the number of edges will be much smaller than N^2, and therefore the dictionaries will perform better, as we can access only the elements they contain).

4.1.3.2 Implementation of Creating Colored Dependency Graph from CDS Network

We now implement the algorithm to create a *Colored Dependency Graph* from a *CDSNetwork*. [26] provide a definition of the *Colored Dependency Graph* $CD(X)$ from a *CDS Network* X:

Definition 8. Colored Dependency Graph Let X be a *CDS Network*, the *Colored Dependency Graph CD(X)* is a graph with N nodes, and green and red edges constructed as follows:

A. For each $i, j \in N$, if $L_{ij} > 0$ or c_{ij}^k for any $k \in N$, add a green edge from $i \to j$

B. For each $i,k \in N$ if $c_{i,j}^k > 0$ for any $j \in N$, add a green edge from $k \to j$

C. For each $j,k \in N$, if j has a naked CDS position toward k, then add a red edge $k \to j$

Performing each of the steps separately would be highly inefficient, as it would require $O(N^3)$ work for each of the separate steps, making it $O(3N^3)$, which simplifies to $O(N^3)$, but in practice is generally worse. Instead, in the alogrithm we implement, we follow the provided rules, but initialise the full graph in just one loop through i, j, k, still providing a worst case run time complexity of $O(N^3)$, but being faster in practice.

In theory, the first two steps in initialising the graph create no issues doing it in one pass, as we can just check for existence of CDS contracts and direct debt in the same iteration, and add green edges for reference entities to creditors and from debtors to creditors. The problem arises when for each $j,k \in N$ we have to check if j has a naked CDS position towards k. Recall j has a naked CDS position towards k if $\sum_{i=1}^{N} c_{ij}^k > L_{kj}$. For each

$k, j \in N$, we store the direct debt L_{kj}, and then loop through all $i \in N$, summing the total value of CDS contracts $\sum_{i=1}^{N} c_{ij}^{k}$, while simultaneously adding green edges according to the first two rules. We finally compare the values of direct debt L_{kj} and total CDS contracts , $\sum_{i=1}^{N} c_{ij}^{k}$, to determine if we need to add a red edge or not. We provide pseudo-code for this algorithm in 4. We join the sets of *redEdges* and *greenEdges* with Python's set union function for each of the keys of the dictionaries, to form one dictionary of *allEdges* which is used in other algorithms.

Algorithm 4 Colored Dependency From CDS Network

```
function GetColoredDependencyFromCDS(network)
    c ← network.c
    L ← network.L
    N ← network.N
    redEdges ← defaultdict(set())
    greenEdges ← defaultdict(set())
    for k in range(N) do
        for j in range(N) do
            debt_k_j ← L[k][j]
            if debt_k_j > 0 then
                greenEdges[k].add(j)
            end if
            CDS_all_j_k ← 0
            for i in range(N) do
                CDS_i_j_k ← c[i][j][k]
                if CDS_i_j_k > 0 then
                    CDS_all_j_k+ = CDS_i_j_k
                    greenEdges[i].add(j)
                    greenEdges[k].add(i)
                end if
            end for
            if debt_k_j < CDS_all_j_k then
                redEdges[k].add(j)
            end if
        end for
    end for
end function
```

4.2 Restricting Colored Dependency Graphs to find Clearing Payments

In this section, we study the restrictions for Colored Dependency Graphs provided in [25]. The different restrictions provide different guarantees for the ability to compute clearing payments for the *CDS Networks*, which we explain under each of the subsections.

For each of the restrictions, we provide our algorithm implementation to check if the restriction holds for the specific *CDS Network*. We then provide our implementation of the algorithm to compute the *Clearing Recovery Rate Vector* when that restriction is proven to hold for the network. Recall that transforming a *Clearing Recovery Rate Vector* r^* into a *Clearing Payment Vector* p^* can be done simply by: $p_i^* = \sum_{j=1}^{N} p_{ij}(r^*)$

4.2.1 Condition I: Acyclic Graph

The first condition provided is that the *Colored Dependency Graph* is *acyclic*. In this context, *acyclic* refers to the non-existence of a directed cycle formed by edges of any colour (any mixture of red and green edges) in the *Colored Dependency Graph*. [26] prove that if this is the case, the *CDS Networks* has a *unique* clearing recovery rate, and therefore this is *maximal*.

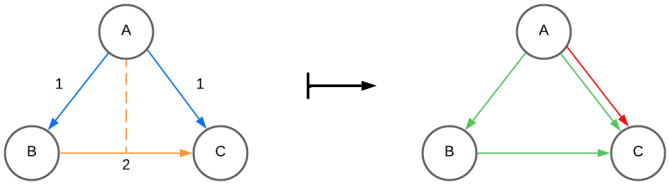


Figure 4.2: Example of *CDS Network* (to the left, ignoring cash flows), for which the *Colored Dependency Graph* (to the right) is *acyclic*. Notice how there is a red edge, as C is short on A, yet the *acyclic* property ensures that there is a unique maximal solution to the clearing payment.

4.2.1.1 Implementation to check for Acyclic Colored Dependency Graph

To check if the graph is acyclic, we implement an algorithm IsCyclic, seen in algorithm (5). The algorithm works with a simple recursive DFS (Depth First Search), which has a worst time complexity of $O(V+E)$, where $V = N$, and $E = length(allEdges)$. IsCyclic stores a set of *seen* nodes, and performs a DFS on all of the nodes that have not yet been added to the set. This DFS fist checks if the node has already been visited (is in the *seen* set), which would imply there is a cycle, at which point it would return True. If the node is not in *seen*, it is added to the set, and the DFS function is recursively called on all the neighbours of that node from the $allEdges$ dictionary.

4.2.1.2 Computing Clearing Payments for Networks with Acyclic Colored Dependency Graphs

Once we have identified that for a *CDS Network* X, *CD(X)* is *acyclic*, we can compute a clearing payment that is *maximal* and *unique*. [26] provide a proof for this, and we adapt this proof to represent the steps our algorithm takes in solving the clearing problem. The steps to find the recovery rate vector r are the following:

A. Create a set *topologicalSet* of the indices of the banks in X sorted topologically based on *CD(X)*. When there is an edge $i \to j$ in $CD(X)$, $i \leq j$. This is possible because there is no cycle in $CD(X)$.

Algorithm 5 IsCyclic

```
function ISCYCLIC(N, ColoredDependencyGraph)
    seen ← ∅
    for node in range (N) do
        if node not in seen then
            if DFS(node) = True then
                return True
            end if
        end if
    end for
    return False
end function
```

B. Intuitively, if the network is acyclic, then the first node in the topological sort has no incoming edges. We can therefore calculate the recovery rate vector r_1 by setting it to the constant function

$$r_1 = \phi_1 = \begin{cases} 1 & \text{if } a_1 \geq l_1 \\ \frac{a'_1}{l_1} & \text{otherwise.} \end{cases}$$

C. Iterate over the elements in *topologicalSet* in order. At each step, set $r_i := \phi_i(r_1, ..., r_{i-1})$, where $r_1, ..., r_{i-1}$ have already been computed.

We implement an algorithm ACYCLICCDSSOLVER that performs the above, shown in algorithm (6). We use Graphlib's TOPOLOGICALSORTER [12] to topologically sort the nodes from the *Colored Dependency Graph* based on the dictionary $allEdges$. The function GETPHIIFORR performs the calculation for $\phi_i(r)$. We provide our implementation in algorithm (7).

Algorithm 6 AcyclicCDSSolver

```
function ACYCLICCDSSOLVER(CDSNetwork, ColoredDependencyGraph)
    sortedNodes ← TOPOLOGICALSORT(ColoredDependencyGraph.allEdges)
    r ← [1, ..., 1]
    for i in range length(sortedNodes) do
        r[i] ← GETPHIIFORR(r[0 : i])
    end for
    return r
end function
```

4.2.2 Condition II: Green Core Systems

The next condition provided by [26] is that the *Colored Dependency Graph* is a *Green Core System*.

Definition 9. Green Core System A *CDS network X* is called a *Green Core System* if in $CD(X)$, all banks with an incoming red edge have no outgoing edge. The set of these banks is called the *leaf set*, and the set of all other banks not in the leaf set, the *core set*.

Algorithm 7 GetPhiIForR

function GETPHIIFORR(i, r, CDSNetwork)
    $assets\_i \leftarrow CDSNetwork.e[i]$
    $liabilities\_i \leftarrow 0$
    for j **in range** $CDSNetwork.N$ **do**
        $liabilities\_i \leftarrow liabilities\_i + CDSNetwork.L[i][j]$
        $assets\_i \leftarrow assets\_i + r[j] \cdot CDSNetwork.L[j][i]$
        for k **in range** $CDSNetwork.N$ **do**
            $liabilities\_i \leftarrow liabilities\_i + (1 - r_k) \cdot CDSNetwork.c[i][j][k]$
            $assets\_i \leftarrow assets\_i + r[j] \cdot ((1 - r_k) \cdot CDSNetwork.c[j][i][k])$
        end for
    end for
    if $assets\_i \geq liabilites\_i$ **then**
        return 1
    else
        $discounted\_assets\_i \leftarrow CDSNetwork.\beta \cdot (assets\_i - CDSNetwork.e[i]) + CDSNetwork.\alpha \cdot CDSNetwork.e[i]$
        return $\frac{discounted\_assets\_i}{liabilities\_i}$
    end if
end function

This condition is enough to ensure that a solution always exists that is best for the banks at the *core*, as proven in [26]. This property implies that *CDS Networks* with no *naked CDS* will *always* have a solution that is maximal for *all* the banks in the network. This is because in the absence of *naked CDS*, there are no red edges in the *Colored Dependency Graph*, implying that the graph is a *Green Core System*, and since no node has an incoming red edge, all nodes are in the *core* set.

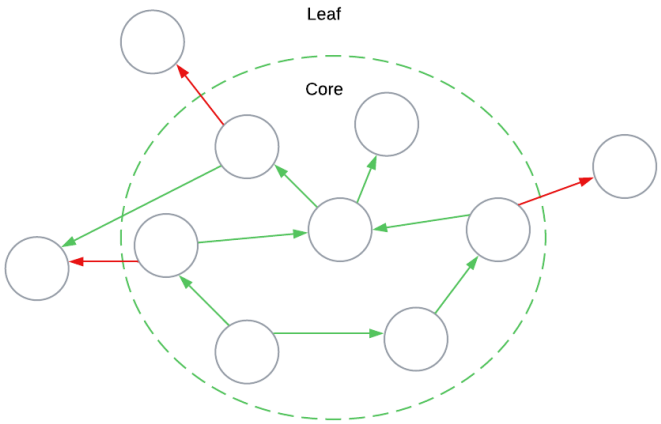


Figure 4.3: Example of a *Green Core System Colored Dependency Graph*. Inside the circle delimitted by the green dotted line are the nodes in the *core set*, and outside, the nodes in the *leaf set*. Note that nodes in the *leaf set* can have incoming green edges too, but no outgoing edge of any kind.

4.2.2.1 Implementation to check for Green Core Systems

We implement an algorithm ISGREENCORE to efficiently find if a *Colored Dependency Graph* is a *Green Core System*. The algorithm loops through the values (referring to values in key:values for the dictionary) in the *redEdge* dictionary, and checks if each value is a key for the *allEdges* dictionary. If it is the case, it means that the nodes has outgoing edges and therefore the *Colored Dependency Graph* is not a *Green Core System*, so we terminate the algorithm. Otherwise, we add this node to the *leaf set*. If the algorithm does not find a node with an incoming red edge and outgoing edges, it creates a *core* set made up of all the nodes that are not part of the *leaf set*.

Algorithm 8 IsGreenCore

```
function ISGREENCORE(N, ColoredDependencyGraph)
    redEdges ← ColoredDependencyGraph.redEdges
    allEdges ← ColoredDependenctGraph.allEdges
    incomingRedEdges ← SUM(redEdges.values())
    for node in incomingRedEdges do
        if node in allEdges then
            return False
        end if
    end for
    return True
end function
```

4.2.2.2 Computing Clearing Payments for Networks with Green Core System Colored Dependency Graphs

The algorithm used to compute clearing payments in these type of restricted networks is an iterative algorithm, similar to those described in (**??**), with the function ϕ for *CDS networks*.

Recall the definition of a *clearing recovery rate vector* r^* for *CDS Networks* as a fixed point of the function ϕ defined by:

$$\phi_i(r) = \begin{cases} 1 & \text{if } a_i(r) \geq l_i(r) \\ \frac{a'_i(r)}{l_i(r)} & \text{otherwise.} \end{cases}$$

The idea behind the algorithm is simple: construct a sequence of vectors (r^n) defined by $r^0 = \{1, ..., 1\}$, and $r^{n+1} = \phi(r^n)$. We recursively apply the function on the previous output, until we find that the values are the same, at which point $r^n = r^{n+1} = r^*$.

The proof that this recovery rate vector is *maximal* for the nodes in the *core* of the *green core system*, and that the iteration sequence converges to the clearing recovery rate vector is provided in [26]. We now provide an implementation of the ITERATIVEGREENCORESOLVER as seen in algorithm (9). This algorithm finds the clearing recovery rate vector as discussed above, and is used when we find that a *CDS Network, X* has a *Colored Dependency Graph*, *CD(X)* which is a *Green Core System*, checked with the ISGREENCORE algorithm.

Algorithm 9 IterativeGreenCoreSolver

```
function ITERATIVEGREENCORESOLVER(CDSNetwork)
    r ← [1, ..., 1]
    for i in range CDSNetwork.N do
        newR ← GETPHIFORR(i, r, CDSNetwork)
    end for
    while r ≠ newR do
        r ← newR
        for i in range CDSNetwork.N do
            newR ← GETPHIFORR(i, r, CDSNetwork)
        end for
    end while
    return r
end function
```

4.3 Theoretical Worst Case Time Complexities of Implementations

In this section we look at the theoretical worst case time complexities of our implementations for each of the restricted network types. For each of them, we look at the time complexity of of checking if the restriction of the network is in place, and then solving for the recovery rate vector. We ignore generating the *Colored Dependency Graph* with the GETCOLOREDDEPENDENCYFROMCDS function, as it is common for both restrictions, and has a worst case time complexity of $O(N^3)$ in both cases.

If we had more time, it would be interesting to experiment to see if GETCOLOREDDEPENDENCYFROMCDS performs better when one of the restrictions is in place.

4.3.1 Condition I: Acyclic Graph

We first look at the algorithm IsCyclic, which checks if the restriction is in place for the network. This algorithm performs a simple DFS in $O(N+E)$, where E is the number of values in the $allEdges$ dictionary.

Once we check that the restriction is met, we look at ACYCLICCDSSOLVER. The first thing the algorithm does is run a topological sort on the $allEdges$ dictionary. Topological sorts themselves can be built via depth first searches or breadth first searches [23], making the complexity of this operation $O(N+E)$. The algorithm then loops through the topologically sorted nodes for $i \in N$. At each iteration, the algorithm calls GETPHIIFORR, passing in a vector r of dimensions i. GETPHIIFORR has a runtime of $O(N^2)$, as it loops through all $(j,k) \in N^2$. This means that the overall time complexity of identifying the restriction, and solving for the clearing recovery rate vector would be: $O(N+E)+O(N)\cdot O(N^2) = O(N^3+E)$

4.3.2 Condition II: Green Core System

For Green Core Systems, we first look at IsGreenCore, to check if the restriction is met. The algorithm loops through all the nodes with an incoming red edge. Let R be the total number of nodes in the values of the $redEdges$ dictionary. Then, for each iteration, it checks if the node is in the $allEdges$ dictionary. This is done in $O(1)$ time. So the overall time complexity for this algorithm is $O(R) \cdot O(1) = O(R)$.

IterativeGreenCoreSolver, the first loop goes through $i \in N$, and runs GetPhiIForR for each i. As explained for condition I, this implies a time complexity of $O(N^3)$. Then, the algorithm has a while loop, and runs GetPhiIForR for each i, at each iteration, until $r \neq newR$. Similar to the discussion of the *Iterative Solver* for *Debt Only Financial Networks* in section (3.3.2), this is an unbounded algorithm, and we cannot provide an asymptotic worst case complexity for it.

The experimentation, which we leave as future work is more interesting here, especially after observing that for *Debt Only Financial Networks*, the performance of the *Iterative Solver* is the least optimal by a large margin. We think experimenting with the IterativeGreenCoreSolver to see if the performance resembles that of the *Iterative Solver* is interesting, as if this is the only possible algorithm for computing clearing payments in this type of restricted networks, when the number of nodes N gets large, computation can become highly inefficient.

Chapter 5

Conclusions

Throughout this thesis, we have provided explanations of different types of financial networks, and the algorithms to compute clearing payments for said networks, based on available research. We found understanding the problem within the different contexts of financial networks was a very challenging task, let alone adapting the notations and explaining them in a constructive way, to facilitate understanding. Of course, due to the landscape of this topic, we did not expect to develop a novel algorithm or discover a new way to compute clearing payment vectors for networks where it is seen computationally unfeasible to do so.

We have however successfully implemented algorithms to compute clearing payments in the three settings we sought out to for this thesis. Even though the theory behind the algorithms was provided, efficiently implementing subroutines for each of the algorithms through the choice of data structures and implementation details has proven to be interesting.

Furthermore, through our experimentation of algorithms for *Debt Only Financial Networks*, in *meaningful networks*, we have identified that the *Linear Solver* is the superior algorithm in terms of runtime, despite having a worst time complexity that is theoretically not polynomial. To our knowledge, we are the first paper to identify the superiority of this algorithm in the context of this problem as opposed to the bounded algorithms such as the *Fictitious Default Solver* or the *Greatest Clearing Vector Solver*.

5.0.1 Practical Relevance

We now would like to have a word on the practical relevance of our thesis. We have provided implementations for which we have tried to optimise the subroutines as much as possible, yet we realise this is far from perfect. Our implementations can be used to understand the algorithms, yet in practice if a clearing entity were to use any of these types of algorithms as part of a clearing process, they would need to be implemented in a much more efficient programming language, such as C.

We are aware that similar tools to some of the algorithms implemented, such as the *Linear Solver* have been used to clear financial networks, such as in Kuwait's al-Manakh

stock market crash in August 1982 [8]. They modelled the financial networks similar to *Debt Only Financial Networks*, and implemented linear programming algorithms to clear the network.

For *CDS Networks*, the tools implemented in this thesis have become, and will continue to become more relevant. [29] explain how the unregulated use of CDS contracts in a speculative manner was a big factor in the 2008 financial crisis. The speculative way of using them was through *naked* CDS contracts, essentially betting on the reference entity defaulting. Since then, in Europe for example, since 2011, there has been a ban of *naked* CDS contracts [1]. With our implementation of ITERATIVEGREENCORESOLVER, we are effectively able to solve *any CDS Network* without *naked* CDS contracts, as they are intuitively Green Core Systems (no naked CDS imply no red edges in the *Colored Dependency Graph*).

5.0.2 Limitations and Future Work

Due to the nature of the research in this paper, we have provided algorithms for a breadth of networks, making it hard to fine-tune the efficiency of each algorithm to perfection. Furthermore, there are many possible extensions to the models we consider that are very interesting to study, and extend the model to implement algorithms for.

5.0.2.1 Extensions to Models

There are many way our models in this thesis can be extended. The first and most clear one would be to implement algorithms to identify the final and least restrictive constraint from [26] for *CDS Networks*, and compute clearing payments for those networks. The constraint is that the Colored Dependency Graph contains no cycles with a red edge between nodes. To tackle the issue of implementing this, we suggest using Tarjan's algorithm for finding strongly connected components in $O(V + E)$ worst-case time complexity [10], and then performing a DFS which checks if a red edge leads to the same strongly connected component, in which case a cycle containing a red edge would be present.

We would also suggest changing the way of modelling bankruptcy costs from the current factors α and β, as suggested by [24], into modelling bankruptcy costs as functions, as seen in [24]. These can be functions of the value of the banks previous to bankruptcy, and can be made to maintain the linearity properties that allow to compute the clearing payment vectors, making it a somewhat straight forward extension to our models. This could be done to better encompass fixed costs associated with bankruptcy, and the value of the bank that is lost upon bankruptcy.

Lastly, we suggest expanding the setting of the networks to allow for banks to have equity in other banks in the network. This setting has been studied by [9] with regards to the effects of cascading of failures in the network. We suggest implementing algorithms to compute clearing payments with the setting provided by them, where banks can have cross-holdings of others, and perhaps extending such model to include CDS contracts in a restricted manner if possible.

5.0.2.2 Experimentation

The first thing we suggest is a thorough experimentation with the algorithms implemented for *CDS Networks*. With more time, this would be the first priority for us. Generating *meaningful CDS networks* is a hard task, due to the three way relations in contracts. [6] study CDS transaction data, and suggest that CDS Networks are highly interconnected. This is a good starting point into attempting to simulate *CDS Networks*, yet extensive research is required to produce something that may even resemble a real network. We suggest that for each restriction provided by [26], networks are generated that satisfy it, and the algorithms are run to see performances, based on number of nodes, or interconnectedness of the network to name a few possible experiments.

For *Debt Only Networks*, and *Debt and Bankruptcy Cost Networks*, we have restricted our experimentation to solely looking at the performance of algorithms by changing the size of the networks N. Firstly, we only experiment with small number of nodes, up to $N = 400$. Even though we find the *Linear Solver* seems to be superior in *meaningful networks* up until that number of nodes, it can be interesting to see what happens for large N. With more time, and a more powerful machine, we would like to see if even for *meaningful networks*, the *Linear Solver* struggles more than the *Greatest Clearing Vector Solver* or the *Fictitious Default Solver* for very large values of N due to the unbounded nature of the algorithm. Also, changing other variables such as the magnitudes of liabilities and equities, and seeing their effect can also be an interesting extension.

Lastly, we suggest experimentation with dependant variables other than runtimes of algorithms. We realise that this is not the only important consideration, and other variables such as memory usage may be interesting to investigate, and take into account when implementing the algorithms.

Bibliography

[1] Regulation on short selling and credit default swaps - frequently asked questions.

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

[3] Encyclopaedia Britannica. Linear programming | definition facts | britannica, 2019.

[4] Julian Caballero. Banking crises and financial integration: Insights from networks science. *Journal of International Financial Markets, Institutions and Money*, 34:127–146, 2015.

[5] Giuseppe C Calafiore, Giulia Fracastoro, and Anton V Proskurnikov. Control of dynamic financial networks. *IEEE Control Systems Letters*, 6:3206–3211, 2022.

[6] Marco D'Errico, Stefano Battiston, Tuomas Peltonen, and Martin Scheicher. How does risk flow in the credit default swap market? *Journal of Financial Stability*, 35:53–74, 2018. Network models, stress testing and other tools for financial stability monitoring and macroprudential policy design and implementation.

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

[8] A. A. Elimam, M. Girgis, and S. Kotob. A solution to post crash debt entanglements in kuwait's al-manakh stock market. *Interfaces*, 27(1):89–106, Feb 1997.

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

[10] Rafael Engibaryan. Finding strongly connected components: Tarjan's algorithm | baeldung on computer science, Apr 2023.

[11] Kousha Etessami. Tutorial: Complexity of equilibria and fixed points: Fixp, fixp, and linear-fixp(= ppad) (and some associated open problems). 2020.

[12] Python Software Foundation. graphlib — functionality to operate with graph-like structures.

[13] Python Software Foundation. time — time access and conversions — python 3.7.2 documentation, 2000.

[14] Galina Hale. Bank relationships, business cycles, and financial crises. *Journal of International Economics*, 88(2):312–325, 2012. NBER Global.

[15] Julian Hall and Karen McKinnon. Hyper-sparsity in the revised simplex method and how to exploit it. *Computational Optimization and Applications*, 32(3):259–283, Dec 2005.

[16] Robert Harper. *Tarski's Fixed Point Theorem for Power Sets* *. 2020.

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

[18] Q. Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method - mathematical programming computation. *SpringerLink*, Dec 2017.

[19] Eun-Jin Im. Optimizing the performance of sparse matrix-vector multiplication, Jun 2000.

[20] Stavros D Ioannidis, Bart de Keijzer, and Carmine Ventre. Strong approximations and irrationality in financial networks with financial derivatives. *arXiv preprint arXiv:2109.06608*, 2021.

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

[22] Jiří Matoušek and Bernd Gärtner. *Understanding and Using Linear Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[23] Chaoyi Pang, Junhu Wang, Yu Cheng, Haolan Zhang, and Tongliang Li. Topological sorts on dags. *Information Processing Letters*, 115(2):298–301, 2015.

[24] Leonard CG Rogers and Luitgard AM Veraart. Failure and rescue in an interbank network. *Management Science*, 59(4):882–898, 2013.

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

[26] Steffen Schuldenzucker, Sven Seuken, and Stefano Battiston. Default ambiguity: Credit default swaps create new systemic risks in financial networks. *Management Science*, 66(5):1981–1998, 2020.

[27] Troy Segal. Syndicated loan, 2019.

[28] Joseph Stiglitz. Honorary lecture by joseph e. stiglitz on climate change and financial complexity, Jan 2016.

[29] René M Stulz. Credit default swaps and the credit crisis. *Journal of Economic Perspectives*, 24(1):73–92, Feb 2010.

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

[31] Jerold B Warner. Bankruptcy costs: Some evidence. *The journal of Finance*, 1977.