Imperial College London

Department of Mathematics

Liquidity Saving Mechanisms and Mixed Integer Linear Programming

Author: Zhiyi XING (CID: 02242064)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2022-2023

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Jordan Cambe. I am so lucky to do my graduating thesis with Jordan. He really helps me a lot through weekly discussions and detailed comments. Additionally, I would like to thank my internal supervisor, Dr. Pietro Siorpaes, who helped me with the formatting of this thesis and kept everything on time.

Finally, I would like to show my deepest gratitude to my family for enabling me to take the master's degree in the Mathematics and Finance program at Imperial College London. Their unconditional support gives me the opportunity to expand my horizons and improve myself.

Abstract

With the development of interbank payment systems, in addition to the pure Real Time Gross Settlement (RTGS) system, Liquidity Saving Mechanisms (LSM) have been introduced to reduce the liquidity needs of payment system participants. LSMs allow payments to be queued and periodically offset each other. Such a problem, that maximizes the total value of settled payments with limited initial balances, has been referred to as the Bank Clearing Problem (BCP). Several algorithms have been proposed to approximately solve this problem. In this thesis, we propose to formulate the BCP as a mixed integer linear programming (MILP) problem and solve it using the Gurobi solver. We then simulate different scenarios of synthetic payment queues based on different parameter settings and compare the performances of our algorithm with two popular algorithms in the LSM literature. We find that our algorithm tends to outperform the two other methods. It also presents three advantages over them: (1) it provides information on the optimality of the solution, which is not possible with other methods; (2) it makes it easy and flexible to change the objective function to meet different optimizing requirements; (3) it can provide optimal solutions as a benchmark for payment system operators to compare with their own system solutions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An interbank payment system consists of some participants (banks)[1] with some payments among them. Every participant has an initial balance and every payment can be settled with enough balance, rejected due to insufficient balance, or delayed (if the system allows, payments can be moved to a queuing system, and settled or rejected later).

In the past few decades, payment systems have rapidly developed and several different types of systems were introduced or replaced, mainly because of the technological innovation of the computing system, the structural changes in participating banks and the evolution of central bank policies [1]. For example, in the Federal Reserve's Fedwire Funds Transfer System (Fedwire), the total value of payments in the system increased from about 100 trillion USD (approximately 15 times the GDP of the U.S.) in 1985 to more than 500 trillion USD (more than 75 times of GDP in the U.S.) in 2005 [2]. Historically, payment systems have developed from Deferred Net Settlement (DNS) systems to Real-Time Gross Settlement (RTGS) systems, and then Liquidity Saving Mechanisms (LSM) were also introduced for further improvement.

Typically, interbank payment systems can be classified as net settlement systems and gross settlement systems, in terms of the computing basis of settlement. For example, a net basis system considers the netting among multiple payments while a transaction-by-transaction basis system settles or rejects each payment individually. Moreover, a deferred settlement system allows the payments to wait in the queue and be settled later while a real-time settlement system decides whether to settle or reject a payment immediately based on whether the current balance is enough or not [3].

A DNS system is a deferred settlement system, and payments are settled simultaneously on a net basis at specific time steps. To reduce the settlement risks that exist in the DNS systems, many central banks started to implement RTGS systems, where the payment is settled in real-time without netting and will be rejected immediately if there is not enough balance for the sender bank to pay for the payment. By 2006, the RTGS systems were implemented in 93 out of 174 central banks over the world [1].

However, the RTGS systems reduce the settlement risk at the cost of increased liquidity requirements to ensure the smooth settlement of each payment. For example, if a bank has to pay a large amount of money before it can receive other payments, then it needs to prepare enough liquidity at the beginning to ensure that it can afford the first several payments before it receives payments from other banks. Bech and Soramäki [4] define a phenomenon called "gridlock" that some payments cannot be all settled by time order due to the lack of liquidity but can be all settled simultaneously, and if no such subset exists, the system becomes deadlocked.

Figure 1.1 and 1.2 illustrate examples of a gridlocked system and a deadlocked system respectively [4]. In both examples, the list of payments is the same, i.e. A needs to pay B 15 units, B needs to pay C 20 units and C needs to pay A 10 units. The only difference is the current balance for the three participants. Suppose the time order of the three payments is: A to B, B to C, and C to A. Then none of the three payments can be settled in the RTGS system based on the time order in both examples. However, they can be settled simultaneously in the first example, i.e. gridlocked system, while none of them can be settled simultaneously in the second example, i.e. deadlocked system.

[1]Throughout the report, "a bank" is equivalent to "a participant" in the interbank payment system

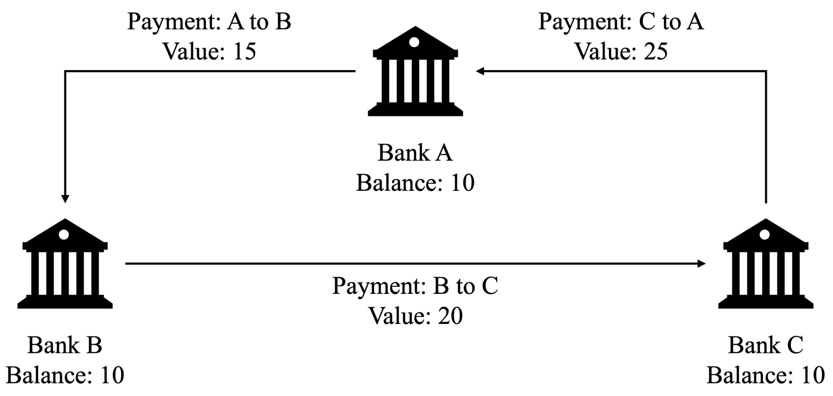


Figure 1.1: Example of Gridlocked System

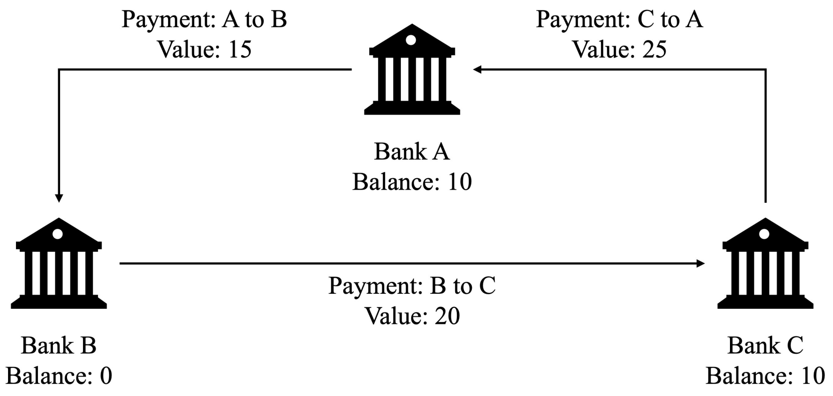


Figure 1.2: Example of Deadlocked System

Therefore, Liquidity Saving Mechanisms (LSM) were introduced as a queuing arrangement to be operated simultaneously with the original RTGS systems. LSMs allow banks to enter the system with less liquidity by delaying some payments and then (partially) offsetting the payments to be sent with the payments to be received simultaneously on a regular basis (e.g. every few minutes). It improves the liquidity requirements in the pure RTGS system by introducing short delays [5]. Usually, LSMs are related to an optimization problem, which maximizes the total value of settled payments with limited liquidity constraints[2] and the order of payments in the queue.

Güntzer et al. [6] defined a discrete optimization problem, the Bank Clearing Problem (BCP), as maximizing the value of settled payments with capacity constraints. Later Bech and Soramäki[7] studied the Gridlock Resolution Problem (GRP), which takes into consideration the order of the payments and adds it as a sequence constraint into the BCP. In this thesis, we formulate the BCP explicitly as a mixed integer linear programming (MILP) problem and find an optimal solution with the Gurobi solver in Python.

To evaluate the performances, we generate synthetic payment data based on different values of 4 parameters to simulate different types of situations in the market, and then implement three algorithms to maximize the total value of settled payments and compare the performance of each algorithm with statistics including the percentage of volume[3]/value settled, efficiency and computing time. Moreover, since the LSMs allow payments to wait, minimization of the total delayed time of the remaining payments in the queue is also a valuable optimization problem to consider. Such a problem can be easily done by our proposed algorithm while difficult to be done by other algorithms.

The remainder of this thesis is structured as follows. Chapter 2 reviews two previous algorithms for solving the BCP and states the motivation for using MILP. In Chapter 3, we introduce several concepts and mathematical approaches to solving the MILP problem. We also formulate the BCP

[2]Limited liquidity for the current queue is also referred to as "initial balance", "capacity", or "deposit".
[3]In this thesis, volume means the number of payments.

into a MILP problem and briefly explain the solving methods. In Chapter 4, we introduce the algorithm we use to generate synthetic payment queues and their related statistical distributions. The results of our simulations under different parameters of queue generation and corresponding solutions to the three algorithms are discussed in Chapter 5. Chapter 6 further discusses the change of objective function for delay time and briefly introduces the multi-objective problem. In Chapter 7, we draw the conclusion of this thesis and provide potential further improvements.

Chapter 2

Literature Review and Motivation

2.1 Current Methods Review

In the past few decades, some research has focused on different types of LSBM Algorithms to reduce the potential risk and improve the interbank payment system. Güntzer et al. (1998)[6] modeled a discrete optimization problem called BCP and proposed five different algorithms to offset payments both bilaterally and multilaterally, in which the fifth algorithm (referred to as Güntzer5 Algorithm in the rest of this thesis) performs better than the others. Bech and Soramäki (2001)[7] proposed an algorithm (referred to as Bech-Soramäki Algorithm in the rest of this thesis) that solving GRP in the RTGS system [3]; Shafransky (2005)[8] proposed a fast heuristic algorithm (S&D approximation algorithm) based on the graph representation of the system. In this chapter, we briefly introduce the Güntzer5 Algorithm and the Bech-Soramäki Algorithm since they are the two most popular algorithms, and compare their performances with that of our proposed algorithm that solves the BCP as a MILP problem later in chapter 5.

2.1.1 Güntzer5 Algorithm

Güntzer et al. [6] defines the BCP that maximizes the total value of settled payments with the constraints of limited liquidity. They propose a relatively simple algorithm to approximately solve the BCP. The key structure of the algorithm consists of three while loops and one post-optimization part.

For each while loop, it will be implemented when there exists any bank with a deficit, i.e. negative balance, and a specific condition for the current system is met. The three while loops continuously activate or deactivate payments until all banks in the system have positive balances and then the algorithm moves to the post-optimization part.

Post-optimization labels each bank by *True* or *False* to detect whether the balance of any bank changes (if the balance changes, then the corresponding bank will be labeled as *True*). This part stops until all banks are labeled as *False* so that it makes settled payments as many as possible.

The pseudo-code of the Güntzer5 Algorithm is shown in Algotirhm 1.

2.1.2 Bech-Soramäki Algorithm

Based on the introduction of gridlock by BIS (1997) [3], Bech and Soramäki[7] propose a simple algorithm to solve the GRP. The main step for this algorithm is that first order payments based on specific requirements, e.g. descending order of the value of the payments. Then, iteratively remove the last payment from a bank's queue until all payments left can be settled by its current balance. Such an order is defined as a sequence constraint for the system operation. With the additional sequence constraints, we can manipulate the optimization variable.

For example, payments can be sorted by value descendingly (resp. ascendingly). Then the payments will be continuously removed from the end of the queue, i.e. from the payment with the smallest (resp. largest) value, until the balance of the bank is non-negative after all payments still in the queue are settled. This particular ordering prioritizes the approximate maximization of the value (resp. volume) of settled payments.

The pseudo-code of the Bech-Soramäki Algorithm is shown in Algorithm 2.

Algorithm 1 Güntzer5 Algorithm

```
Input  : queue, capacity(initial balance)
Output: settled payment, updated balance
Initialize status = 1 for all payments, i.e. activate all payment orders
// In all algorithms and Python codes, we assign 0 to status to represent an
   inactive payment and 1 for an active payment.
Update the balance after settling all payments
while balance of any bank $< 0$ do
    current bank = the bank with the largest deficit
    while current bank balance $< 0$ and $\exists$ a payment to current bank with status = 0 s.t. value $<$
     sender balance do
        list = {payment: receiver = current bank, status = 0 and value $<$ sender balance}
        sublist = {payment: payment $\in$ list and value $>$ current bank deficit }
        if sublist is not empty then
            Set the status of the minimum payment in the sublist to 1
        else
            Set the status of the maximum payment in the list to 1
        end
    end
    while current bank balance $< 0$ and $\exists$ a payment from current bank with status = 1 s.t. value
     $<$ receiver balance do
        list = {payment: sender = current bank, status = 1 and value $<$ receiver balance}
        sublist = {payment: payment $\in$ list and value $>$ current bank deficit}
        if sublist is not empty then
            Set the status of the minimum payment in the sublist to 0
        else
            Set the status of the maximum payment in the list to 0
        end
    end
    while current bank balance $< 0$ do
        list = {payment: sender = current bank and status = 1}
        sublist = {payment: payment $\in$ list and value $>$ current bank deficit}
        if sublist is not empty then
            Set the status of the minimum payment in the sublist to 0
        else
            Set the status of the maximum payment in the list to 0
        end
    end
end
Initialize postoptimization status as True for all banks
while any bank with postoptimization status == True do
    current bank = the first bank with postoptimization status True
    // Banks are sorted in alphabetical order.
    list = {payment: sender = current bank and status = 0, in descending order of value}
    for each payment in list do
        if value $<$ current bank balance then
            Set the status of this payment to 1
            Update the current bank balance as: balance -= value
            Set the postoptimization status of the receiver bank for this payment to True
    end
    Set the postoptimization status of current bank to False
end
Return the payments with status = 1 as optimal settled payments
```

Algorithm 2 Bech-Soramäki Algorithm

```
Input  : queue, capacity(initial balance)
Output: settled payment, updated balance
Initialize status = 1 for all payments, i.e. activate all payment orders
Sort all payments according to the requirement
Update the balance after settling all payments
Set $l_i$ = list of payments (in order) from bank $i$
while balance of any bank < 0 do
    current bank = the bank with the largest deficit and non-empty queue
    Remove the last payment in $l_{currentbank}$
    current bank balance += value of this payment
    balance receiver bank of this payment -= value of this payment
end
Return the remaining payments in queues as optimal settled payments
```

Figure 2.1 gives an example of how the Bech-Soramäki Algorithm works when the order of the payments in the queue is sorted descendingly i.e. large payments first (referred to as sort = 2). The original queue is sorted by time of arrival and the last payment in the queue is removed in each step. Simultaneously, the value of this removed payment is added to the balance. The process stops when the balance is non-negative and current payments in the queue are settled payments.

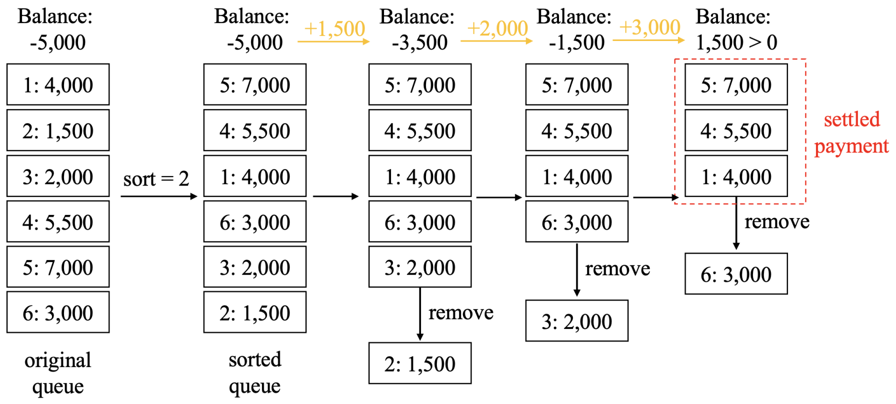


Figure 2.1: Illustration of Bech-Soramäki Algorithm

2.2 Motivation of Using MILP Problem

In this thesis, we propose a new algorithm (referred to as BM Algorithm in this thesis) to explicitly formulate the BCP problem into a 0-1 mixed integer linear problem and solve it with Gurobi. The motivations and advantages of using the BM Algorithm are listed as follows:

1. Performance evaluation: Algorithms proposed before are local search methods, which can only provide the approximating solution and it is impossible to evaluate the goodness of the solution. Using the BM Algorithm, we can exactly read from the log file to get the tolerance setting and the gap between the current solution and the upper bound.

2. Flexibility of target: Algorithms proposed before are specifically defined to maximize the total value of settled payments and impossible to change the optimizing target. Using the BM Algorithm, we explicitly define the whole problem as a MILP problem, which makes it easy and flexible to explore different optimizing targets by simply changing the objective function.

3. Benchmarking: Our BM Algorithm to find an optimal solution is established in an explicitly mathematical way and solve the MILP problem mainly based on the Branch-and-Cut method. Therefore, our BM Algorithm solution can act as a benchmark for other central banks to compare the performances of their own systems with the performance given by our algorithm.

4. Capability for Multi-Objective Optimization: Based on the explicit mathematical expressions for all attributes of the payment, we can formulate the objective functions related to value, volume, delay, etc. In practice, the system can not only find the maximum value of settled payments but also find the maximum settled volume without losing too much for the settled value, which will be further discussed in chapter 6.

Chapter 3

Mixed Integer Linear Programming (MILP)

In this chapter, we introduce the definition of terminologies for a MILP problem and several methods for solving a MILP problem. Most of the methodologies are referenced to Chapter 1 in *Integer Programming* [9] and original articles where the methods were initially proposed[1]. Then we formulate the BCP into a MILP problem by explicitly giving its mathematical expressions and briefly introduce the process for Gurobi solving a MILP problem, which is implemented for our simulations. For simplicity, we assume that any linear programming problem in this chapter has at least one feasible solution.

3.1 Introduction

Definition 3.1.1. A *mixed integer linear programming (MILP)* problem is a problem of the form

$$\begin{aligned} \text{maximize} \quad & cx + hy \\ \text{subject to} \quad & Ax + Gy \le b, \\ & \qquad x \ge 0 \quad \text{integers}, \\ & \qquad y \ge 0 \end{aligned} \tag{3.1.1}$$

where row vectors $c = (c_1, \ldots, c_n) \in \mathbb{R}^n$, $h = (h_1, \ldots, h_p) \in \mathbb{R}^p$, matrices $A = (a_{ij}) \in \mathbb{R}^{m \times n}$, $G = (g_{ij}) \in \mathbb{R}^{m \times p}$ and a column vector $b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \in \mathbb{R}^m$ are the data and column vectors $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{Z}^n_+$, $y = \begin{pmatrix} y_1 \\ \vdots \\ y_p \end{pmatrix} \in \mathbb{R}^p_+$ are variables to be optimized, $n, m, p \in \mathbb{N}$.

Remark 3.1.2. In this report, for simplicity, we use $\mathbb{Z}^n_+$ and $\mathbb{R}^n_+$ to denote the set of non-negative integers and real numbers respectively, i.e. $0 \in \mathbb{Z}^n_+$ and $0 \in \mathbb{R}^n_+$. Also, we use $\mathbb{N}$ to denote the set of nature numbers including 0.

Definition 3.1.3. A *mixed integer linear set* is the set of the form:

$$D := \left\{(x, y) \in \mathbb{Z}^n_+ \times \mathbb{R}^p_+ : Ax + Gy \le b\right\} \tag{3.1.2}$$

i.e. the set of feasible solutions to the MILP problem (3.1.1)

For convenience, we denote (3.1.1) as:

$$P := \max\left\{cx + hy : (x, y) \in D\right\} \tag{3.1.3}$$

[1]Specific reference will be added for these methods when mentioned in subsections.

3.2 Pre-solve

The purpose of pre-solving is to further simplify the original problem by taking advantage of the integrality of variables. Some variables can be calculated to specific values and some rows or columns are likely to be removed.

Example 3.2.1. *Assume that we have a MILP problem as follows:*

$$\begin{aligned}
\text{maximize} \quad & x_1 + x_2 + x_3 + x_4 \\
\text{subject to} \quad & x_1 + x_2 \le 1.5 && (1), \\
& x_1 \ge 1 && (2), \\
& x_1 + x_3 + x_4 \le 6 && (3), \\
& x_1, x_2 \in \mathbb{N}, \quad x_3, x_4 \ge 0 && (4)
\end{aligned}$$

Combining (1),(2) and (4), the only feasible solution for x_1, x_2 is $x_1 = 1$ and $x_2 = 0$ and then inequalities (1) and (2) and be removed and (3) can be revised as $x_3 + x_4 \le 5$. Moreover, the objective function can be written as $x_3 + x_4 + 1$.

Finally, the original MILP problem with 3 inequalities and 4 variables is simplified to a new MILP problem with only 1 inequality and 2 variables as follows, without losing any original feasible solution.

$$\begin{aligned}
\text{maximize} \quad & x_3 + x_4 + 1 \\
\text{subject to} \quad & x_3 + x_4 \le 5, \\
& x_3, x_4 \ge 0
\end{aligned} \tag{3.2.1}$$

3.3 Linear Programming (LP) Relaxation

The main idea for LP relaxation is to remove the integrality constraints of variables and transform the MILP problem into a more general linear programming problem. The optimal value given by the solution of LP relaxation provides an upper bound for the optimal value of the original MILP problem.

Definition 3.3.1. A *natural linear relaxation* R for the set D defined in (3.1.2) is defined by removing the integrality constraint for x as:

$$R = \left\{(x, y) \in \mathbb{R}^n_+ \times \mathbb{R}^p_+ : Ax + Gy \le b\right\} \tag{3.3.1}$$

Definition 3.3.2. A *natural linear programming (LP) relaxation* LP of the MILP problem P defined in (3.1.1) is defined by substituting R for D as:

$$LP := \max\{cx + hy : (x, y) \in R\} \tag{3.3.2}$$

Proposition 3.3.3. *Suppose (x^*, y^*) is the optimal solution of LP with optimal value z^*.Then z^* is the upper bound of the optimal value of the original MILP P.*

Proof. By the definition of natural linear relaxation, $D \subseteq R$ and then any feasible solution (x, y) with value z for original MILP problem P satisfies $(x, y) \in D \subseteq R$. Since the objective function is maximum, we have:

$$z = cx + hy \le \max\{cx + hy : (x, y) \in D\} \le \max\{cx + hy : (x, y) \in R\} = z^*$$

□

3.4 Branch-and-Bound Method

3.4.1 Methodology

Based on the notations in chapter 1 of *Integer Programming* [9], consider a general MILP problem $P = \max\{cx + hy : (x, y) \in D\}$ where $D = \left\{(x, y) \in \mathbb{Z}^n_+ \times \mathbb{R}^p_+ : Ax + Gy \le b\right\}$. Let LP and R be the natural LP relaxation and natural linear relaxation for P and D, respectively. Suppose (x^0, y^0)

is the optimal solution to LP and there exists some element in x that is fractional, i.e. x_i^0 for some $1 \leq i \leq n$.

Define two sets as:

$$D_1 := D \cap \left\{(x,y) : x_i \leq \lfloor x_i^0 \rfloor\right\}, \quad D_2 := D \cap \left\{(x,y) : x_i \geq \lceil x_i^0 \rceil\right\}$$

where $\lfloor x \rfloor$ is the largest integer that smaller than or equal to x and $\lceil x \rceil$ is the smallest integer that larger than or equal to x.

Define the two corresponding sub-problems as:

$$\begin{aligned} P_1 &= \max\left\{cx + hy : (x,y) \in D_1\right\} \\ P_2 &= \max\left\{cx + hy : (x,y) \in D_2\right\} \end{aligned}$$

Proposition 3.4.1. *Suppose two sub-problems P_1, P_2 have optimal values z_1^* and z_2^*. Then the optimal value of P is $z^* = \max\{z_1^*, z_2^*\}$.*

Proof. By the integrality of vector x in D, it is obvious that D_1, D_2 are disjoint and $D_1 \cup D_2 = D$. Suppose the corresponding solutions are (x_1^*, y_1^*) and (x_2^*, y_2^*), then:

$$\begin{aligned} (x_1^*, y_1^*) \in D_1 \quad &\text{and} \quad z_1^* = cx_1^* + hy_1^* \geq cx_1 + hy_1 \quad \forall (x_1, y_1) \in D_1 \\ (x_2^*, y_2^*) \in D_2 \quad &\text{and} \quad z_2^* = cx_2^* + hy_2^* \geq cx_2 + hy_2 \quad \forall (x_2, y_2) \in D_2 \end{aligned}$$

Since $D_1 \cup D_2 = D$, we have:

$$\max\{z_1^*, z_2^*\} \geq cx + hy \quad \forall (x,y) \in D$$

Therefore, the optimal value of P is $z^* = \max\{z_1^*, z_2^*\}$ □

Let LP_1 and LP_2 be the natural LP relaxation of P_1 and P_2, respectively. Let R_1 and R_2 be the natural linear relaxation of D_1 and D_2, respectively.

All different cases are listed as follows:

1. If $R_i = \emptyset$, which implies LP_i is infeasible, then:

$$D_i \subseteq R_i \quad \Rightarrow \quad D_i = \emptyset \quad \Rightarrow \quad P_i \text{ is infeasible}$$

   In this case, we do not need to consider the sub-problem P_i any further and say that P_i is *pruned by infeasibility.*

2. If $R_i \neq \emptyset$ for $i = 1, 2$, then let $\left(x^i, y^i\right)$ be an optimal solution to LP_i with optimal value z^i, $i = 1, 2$. Let l and u be the current best lower bound and upper bound for the optimal value.

   (a) If every element in x^i is an integer, i.e. $\left(x^i, y^i\right) \in D_i$, then $\left(x^i, y^i\right)$ is the optimal solution of P_i and the corresponding optimal value z^i is the updated lower bound, i.e. $l = z^i$. We do not need to consider the sub-problem P_i any further since it is solved already and say that P_i is *pruned by integrality.*

   (b) If there exists an element in x^i is not an integer and $z^i \leq l$, then it is impossible to find a better solution in D_i by proposition 3.3.3 as:

$$z \leq z^* \leq l \quad \forall z \text{ as an optimal value for } P_i$$

   We do not need to consider the sub-problem P_i any further since it is solved already and say that P_i is *pruned by bound.*

   (c) If there exists an element in x^i is not an integer and $z^i > l$, then it is possible to find a better solution in D_i and we need continue to do the branching process for $\left(x^i, y^i\right)$.

3.4.2 Example

Suppose we have a MILP problem as follows:

$$\begin{aligned} \text{maximize} \quad & 5.5x_1 + 2.1x_2 \\ \text{subject to} \quad & -x_1 + x_2 \le 2, \\ & 8x_1 + 2x_2 \le 17, \\ & x_1, x_2 \ge 0 \quad \text{integers} \end{aligned} \tag{3.4.1}$$

In this section, we solve the MILP problem P_0 (3.4.1) to illustrate the above four different cases in the Branch-and-Bound method.

Step 1: Solve the LP relaxation LP_0 of (3.4.1) and get the optimal solution $x_1^0 = 1.3$ and $x_2^0 = 3.3$ with optimal value $z^0 = 14.08$

Step 2: Since the solution of LP_0 does not satisfy the integrality, we create a branch on x_1 by separating the feasible region into two parts with $x_1 \le 1$ and $x_1 \ge 2$ so that we define two sub-problems P_1 and P_2 as follows. The corresponding LP relaxations are denoted as LP_1 and LP_2, respectively.

Sub-problem P_1:

$$\begin{aligned} \text{maximize} \quad & 5.5x_1 + 2.1x_2 \\ \text{subject to} \quad & -x_1 + x_2 \le 2, \\ & 8x_1 + 2x_2 \le 17, \\ & x_1, x_2 \ge 0 \quad \text{integers}, \\ & x_1 \le 1 \end{aligned}$$

Sub-problem P_2:

$$\begin{aligned} \text{maximize} \quad & 5.5x_1 + 2.1x_2 \\ \text{subject to} \quad & -x_1 + x_2 \le 2, \\ & 8x_1 + 2x_2 \le 17, \\ & x_1, x_2 \ge 0 \quad \text{integers}, \\ & x_1 \ge 2 \end{aligned}$$

Step 3: For LP_1, we can solve for the optimal solution $x_1^1 = 1$ and $x_2^1 = 3$ with optimal value $z^1 = 11.8$. Note that all variables are integers so this is a feasible solution to the original MILP problem (3.4.1) and there is no need to continue from this node, which is called *pruned by integrality.*

Step 4: For LP_2, we can get the optimal solution $x_1^2 = 2$ and $x_2^2 = 0.5$ with optimal value $z^2 = 12.05$. Note that x_1 is an integer while x_2 is still fractional and also the optimal value is larger than z^1, which requires further branching on this node. Figure 3.1 shows the solutions to the three LP relaxations on the plane.

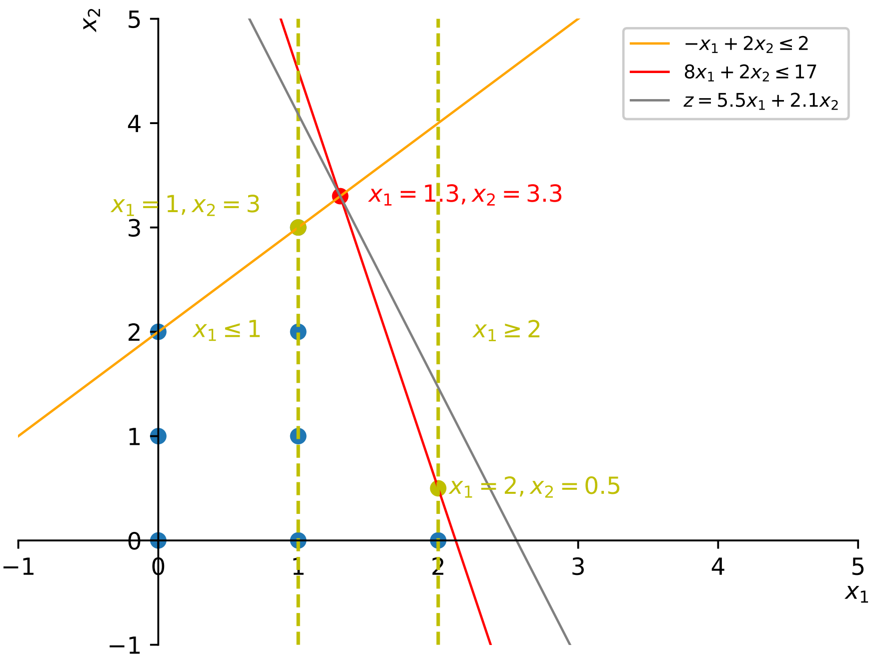


Figure 3.1: LP Relaxation and First Branching

Step 5: By the conclusion in step 4, we continue to create a branch from LP_2 node on x_2 by separating the feasible region into two parts with $x_2 \leq 0$ and $x_2 \geq 1$ so that we define two sub-problems P_3 and P_4 as follows. The corresponding LP relaxations are denoted as LP_3 and LP_4, respectively.

Sub-problem P_3:

$$\begin{aligned} \text{maximize} \quad & 5.5x_1 + 2.1x_2 \\ \text{subject to} \quad & -x_1 + x_2 \leq 2, \\ & 8x_1 + 2x_2 \leq 17, \\ & x_1, x_2 \geq 0 \quad \text{integers}, \\ & x_1 \geq 2, \\ & x_2 \leq 0 \end{aligned}$$

Sub-problem P_4:

$$\begin{aligned} \text{maximize} \quad & 5.5x_1 + 2.1x_2 \\ \text{subject to} \quad & -x_1 + x_2 \leq 2, \\ & 8x_1 + 2x_2 \leq 17, \\ & x_1, x_2 \geq 0 \quad \text{integers}, \\ & x_1 \geq 2, \\ & x_2 \geq 1 \end{aligned}$$

Step 6: For LP_3, the optimal solution is $x_1^3 = 2.2125$ and $x_2^3 = 0$ with optimal value $z^3 = 11.6875$. Note that 11.6875 is less than the optimal value of 11.8 given by LP_1, which is the lower bound for the solution to the MILP problem since it already satisfies the integrality. Therefore, $11.6875 < 11.8$ stops the following process for LP_3 node and this is the case called *pruned by bound.*

Step 7: For LP_4, Figure 3.2 shows that after adding two branching inequalities $x_1 \geq 2$ and $x_2 \leq 0$, the feasible region becomes empty, so that LP_4 is infeasible and the process stops here. This situation is called *pruned by infeasibility.*

Step 8: Note that all the branches are pruned and we have Figure 3.3 containing all the nodes with solution information. Comparing the terminal solution of each branch, it is easy to conclude that the solution to LP_1, i.e. $x_1^1 = 1, x_2^1 = 3$ with value $z^1 = 11.8$, is the optimal solution to the original MILP problem.

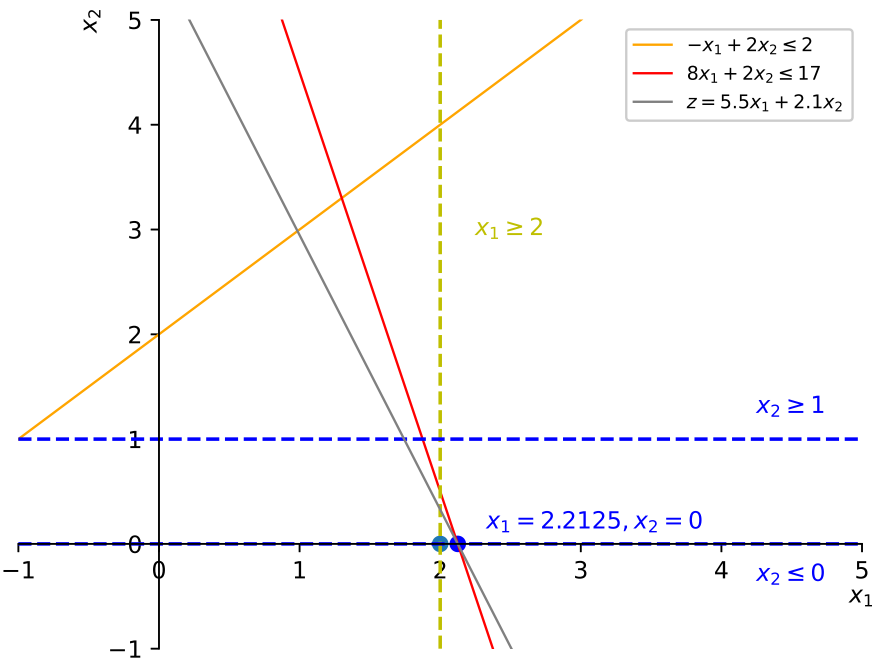


Figure 3.2: Second Branching

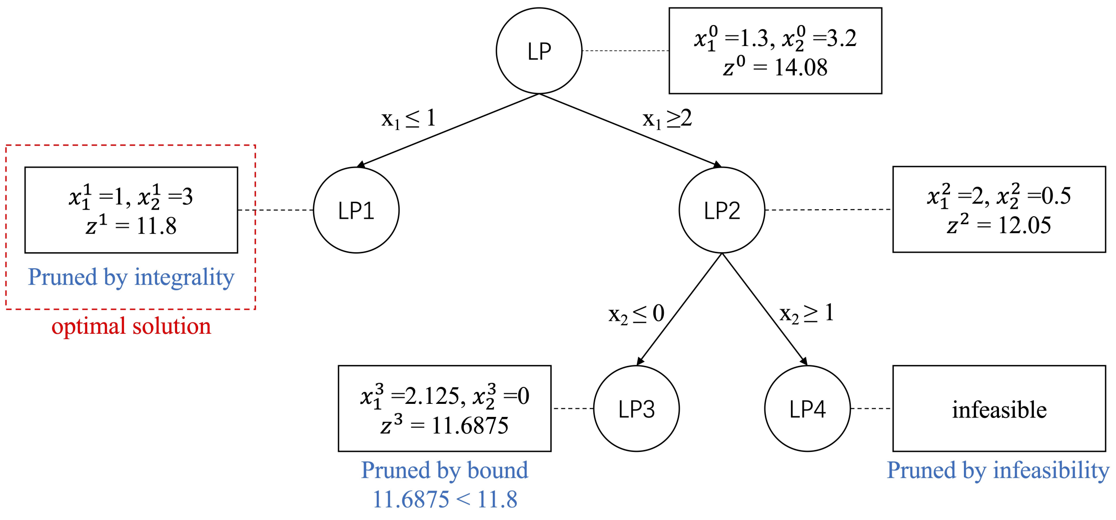


Figure 3.3: Branch-and-Bound Tree

3.5 Cutting Plane Method

The cutting plane method is a fundamental approach to solving a MILP problem and in the past few decades, many different types of algorithms were introduced to generate different cuts such as tableau-based disjunctive cuts (e.g. Gomory cuts, Mixed-Integer-Rounding (MIR) cuts), knapsack cuts (e.g. cover cuts), etc. Marchand [10] summarized several useful cutting planes including Gomory's mixed integer cuts, MIR cuts, lift-and-project cuts, etc.

In this section, we briefly introduce the general methodology of cutting plane methods and three specific generations of cuts, i.e. cover cuts, Chvátal-Gomory cuts and reformulation-linearization technique (RLT), which are most commonly used in our implementations of the BM Algorithm.

3.5.1 Methodology

Consider the same general MILP problem P in section 3.4.1 and suppose $\left(x^0, y^0\right)$ is the optimal solution to LP with optimal value z^0.

Definition 3.5.1. A *cutting plane* separating a point (x^0, y^0) from a set D is defined as a valid inequality $\alpha x + \gamma y \leq \beta$ such that

$$\alpha x + \gamma y \leq \beta \quad \forall (x, y) \in D \quad \text{and} \quad \alpha x^0 + \gamma y^0 > \beta$$

Figure 3.4 gives an illustration of the definition of a cutting plane, where the blue dashed line represents the cutting plane that separates the optimal solution to LP, i.e. the red point, from the integer feasible set.

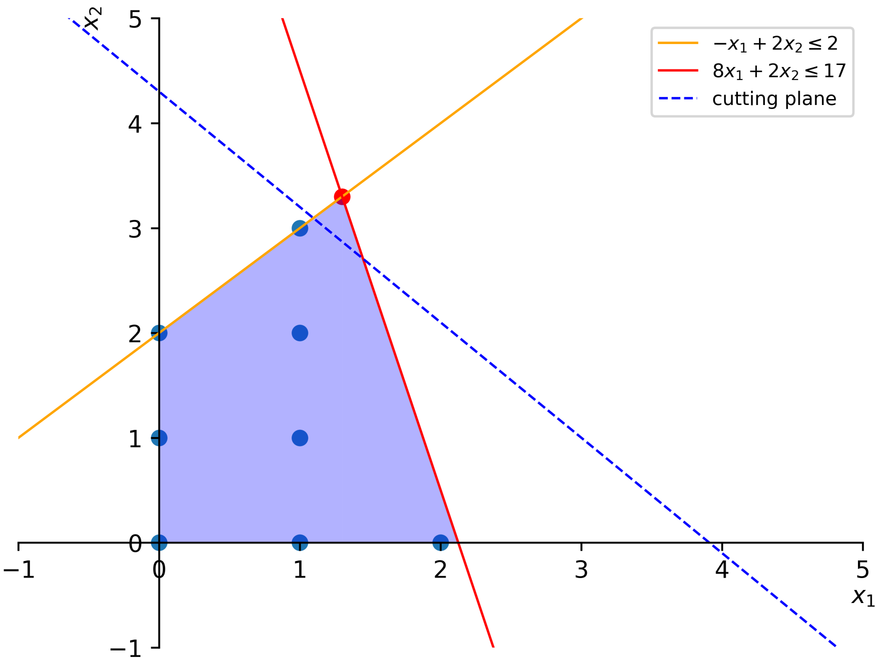


Figure 3.4: Example of a Cutting Plane

The key idea for cutting planes methods is that if $\left(x^0, y^0\right) \in R \backslash D$, then adding a cutting plane separating $\left(x^0, y^0\right)$ from D gives a smaller feasible sub-region

$$D_1 = D \cap \{(x, y) : \alpha x + \gamma y \leq \beta\}$$

i.e. retaining all feasible solutions of P and removing the optimal solution of LP.

Then define the new natural linear relaxation and LP relaxation as:

$$R_1 = R \cap \{(x, y) : \alpha x + \gamma y \leq \beta\} \quad \text{and} \quad LP_1 = \max\{cx + hy : (x, y) \in R_1\}$$

Find the optimal solution $\left(x^1, y^1\right)$ for LP_1 with optimal value z^1. If $\left(x^1, y^1\right) \notin D_1$, then keep adding a new cutting plane seperating $\left(x^1, y^1\right)$ from D_1 to get a new sub-region D_2, the corresponding natural linear relaxation R_2 and LP relaxation $LP_2, \ldots$ Repeat the addition of cutting planes until $\left(x^i, y^i\right) \in D_i$ for some i and then $\left(x^i, y^i\right)$ with value z^i is the optimal solution to the original MILP problem P.

3.5.2 Cover Cut

The cover cut is a set of cover inequalities first proposed by Balas[11] in 1973 and then extensions were discussed and such inequalities were applied to the cutting plane method.

In this section, we discuss the cover cut for a specific type of MILP problem, the 0-1 MILP problem, where all the variables are decision variables, i.e. 0 or 1. Under the same representation, we can write a general 0-1 MILP problem as:

$$P = \max\{cx : x \in D\} \quad \text{where } D = \{x \in \{0,1\}^n : Ax \leq b\} \tag{3.5.1}$$

where $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$.
Suppose the j-th inequality in the constraint satisfies:

$$\sum_{i=1}^{n} a_{ji}x_i \leq b_j \tag{3.5.2}$$

where $a_{ji} > 0$, for all $i = 1, \ldots, n$ and $b_j > 0$

Definition 3.5.2. A subset $C \subseteq \{1, \ldots, n\}$ is a *cover* for (3.5.2) if:

$$\sum_{i\in C} a_{ji} > b_j$$

i.e. only taking all variables with index in C has already exceeded the constraint boundary.

Definition 3.5.3. A cover C for (3.5.2) is *minimal* if for all proper subset Q of C:

$$\sum_{i\in Q} a_{ji} \leq b_j$$

Theorem 3.5.4. *Let $\Gamma = \{C : C$ is the minimal cover for (3.5.2)$\}$ and $|C|$ denote the number of elements in C. Then (3.5.2) is equivalent to the set of inequalities:*

$$\sum_{i\in C} x_i \leq |C| - 1 \quad \forall C \in \Gamma \tag{3.5.3}$$

where each inequality is called a cover inequality of cover C.

Example 3.5.1. *Consider the following MILP problem*

$$\begin{aligned} \text{maximize} \quad & 5.5x_1 + 2.1x_2 \\ \text{subject to} \quad & -x_1 + x_2 \leq 2, \\ & 9x_1 + 10x_2 \leq 17, \\ & \quad x_1, x_2 \geq 0 \quad \textit{integers} \end{aligned}$$

Note that the second inequality $9x_1 + 10x_2 \leq 17$ satisfies that all the coefficients and R.H.S value are positive.

By definition of the minimal cover, it is easy to see that $C = \{1, 2\}$ is the only minimal cover and therefore applying Theorem 3.5.4 gives the cover inequality:

$$x_1 + x_2 \leq 2 - 1 = 1$$

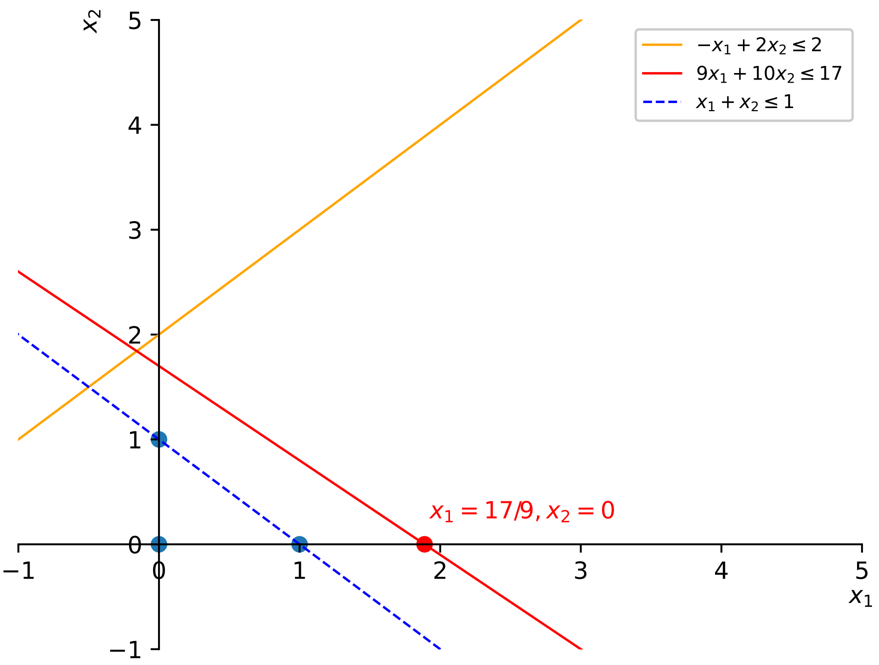


Figure 3.5: Example of a Cover Inequality

3.5.3 Chvátal-Gomory Cut

In 1958, Gomory [12] first proposed a general cutting method for a pure integer programming problem, i.e. a general MILP defined in (3.1.1) with $p = 0$, and then combined with the contributions of Chvátal [13], the improved Chvátal-Gomory inequality was introduced to be a common cut used in many solvers.

Proposition 3.5.5 (C-G inequality)**.** *Let $Ax \le b$ be the linear system of inequalities where x is a non-negative integer vector, i.e. $Ax \le b$ are constraints for a pure MILP problem. Let $u \in \mathbb{R}^m_+$ be a non-negative real vector. Then there exists a valid inequality defined as:*

$$\sum_{i=1}^{n} \lfloor uA_i \rfloor x_i \le \lfloor ub \rfloor \tag{3.5.4}$$

where A_i denotes the i-th row of A, $i = 1, \dots, n$.This inequality is called Chvátal-Gomory inequality (C-G inequality)

Proof. By the definition of the floor, since u and x are non-negative, we have:

$$\sum_{i=1}^{n} \lfloor uA_i \rfloor x_i \le \sum_{i=1}^{n} uA_i x_i \le ub$$

Then since $\lfloor uA_i \rfloor$ and x_i are integers for all $i = 1, \dots, n$, $\sum_{i=1}^{n} \lfloor uA_i \rfloor x_i$ must be an integer so that it must be less than or equal to the largest integer not larger than ub, i.e.

$$\sum_{i=1}^{n} \lfloor uA_i \rfloor x_i \le \lfloor ub \rfloor$$

□

Note that proposition 3.5.5 holds for any non-negative vector u so that corresponding CG-inequalities are valid. However, the plane created by these CG-inequalities may not be the cutting plane for LP relaxation, which depends on the choice of u. Further research and improvement are made such as implementing the simplex method before choosing u.

Example 3.5.2. *Consider the same example MILP problem in (3.4.1)*

$$\begin{aligned} \text{maximize} \quad & 5.5x_1 + 2.1x_2 \\ \text{subject to} \quad & -x_1 + x_2 \leq 2, \\ & 8x_1 + 2x_2 \leq 17, \\ & x_1, x_2 \geq 0 \quad \textit{integers} \end{aligned}$$

Take $u = (0.4, 0.3)$*, we have:*

$$2x_1 + x_2 \leq 5.9$$

By C-G inequality, taking the floor of coefficients and R.H.S, we get a new cut as:

$$2x_1 + x_2 \leq 5$$

The cutting plane is illustrated in Figure 3.6 with the blue dashed line. It shows that this inequality indeed adds a cutting plane that separates the LP optimization solution and retains all feasible integral solutions.

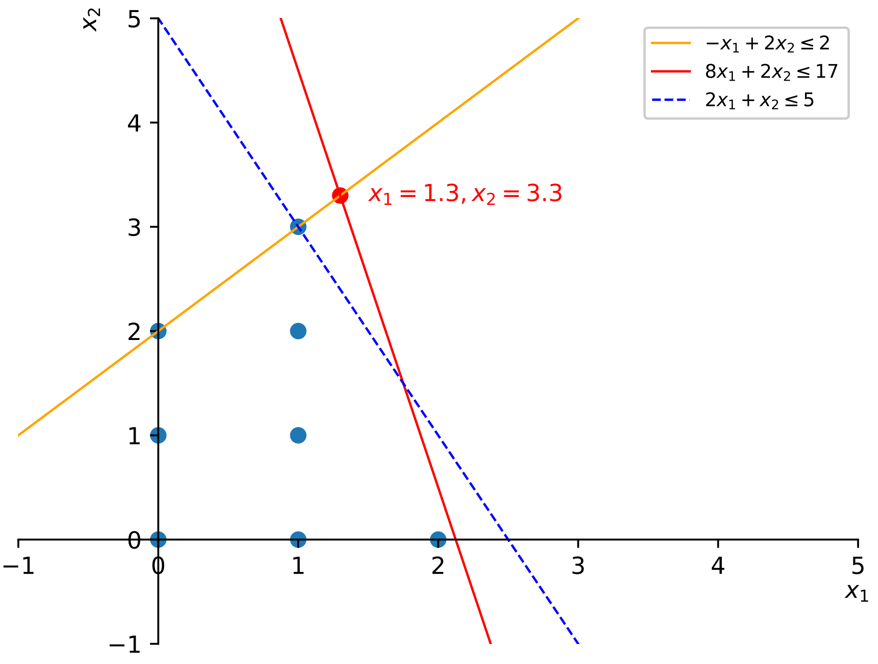


Figure 3.6: Example of a C-G Inequality

3.5.4 Reformulation-Linearization Technique (RLT)

In this section, we also discuss the RLT for the 0-1 MILP problem as (3.5.1). Sherali and Adams [14] first invented the RLT and conclude that higher level of the hierarchy k results in stronger RLT relaxation and optimal RLT can be retrieved when $k = n$. Franklin et al. [15] review the algorithm and summarize a specific process for the 0-1 MILP problem, which adds the cutting plane to the continuous relaxation defined as below.

Definition 3.5.6. The *continuous relaxation* of a 0-1 MILP problem as (3.5.1) is a problem that relaxes the binary constraint of the variable to the interval [0,1] as:

$$CR = \max \{cx : x \in R_C\} \quad \text{where } R_C = \{x \in [0,1]^n : Ax \leq b\} \tag{3.5.5}$$

For any given $k \in \{1, \ldots, n\}$, the level-k RLT relaxation has two steps as follows:

- Reformulation step: Construct a system K of valid polynomial inequalities of degree $k+1$. Let S, T be disjoint subsets of $\{1, \dots, n\}$. Define:

$$J(S,T) = \prod_{i \in S} x_i \prod_{i \in T} (1 - x_i)$$

  - If $|S| + |T| = k+1$, then add the inequality $J(S,T) \geq 0$ into the system K.
  - If $|S| + |T| = k$, then add the inequality $J(S,T)(b_j - \sum_{i=1}^{n} a_{ji} x_i) \geq 0$ into the system K, $j = 1, \dots, m$.

- Linearization step: Substitute new variables for monomials of degree > 1
  - Expand the left-hand side of each inequality in the system K to the format as a weighted sum of distinct monomials
  - Substitute x_i for x_i^r for all $i = 1, \dots, n$ and $r = 2, \dots, n+1$.
  - For each inequality in the system K, if $2 \leq |S| \leq \min\{k+1, n\}$, then define $y_S = \prod_{i \in S} x_i$ be a new binary variable and replace the corresponding product term in each inequality

  Now we get a system of linear inequalities with binary variables and each inequality has the format as:

$$\sum_{i=1}^{n} \alpha_i x_i + \sum_{2 \leq |S| \leq \min\{k+1,n\}} \beta_S y_S \leq \gamma \tag{3.5.6}$$

  where α_i and β_S are coefficients computed by reorganizing the terms of the inequality for some constant γ.
  The system with inequalities as (3.5.6) provides the RLT cutting plane to solve the 0-1 MILP problem.

3.6 Branch-and-Cut Method

In practice, we combine the Branch-and-Bound approach in section 3.4 and the cutting plane methods in section 3.5 together to more efficiently find the optimal solution to the MILP problem, since the cutting plane method can provide tighter upper bound for each problem, which is important for pruning the Branch-and-Bound Tree.

In chapter 1 of *Integer Programming* [9], a formal algorithm of Branch-and-Cut method is constructed as follows:
Let N_i be the node representing problem P_i,$\mathcal{L}$ be the set of non-pruning nodes, $\underline{z}$ be the current lower bound and (x^*, y^*) be the optimal solution.

Step 1: Initialization: Set $\mathcal{L} = \{N_0\}$, $\underline{z} = -\infty$ and $(x^*, y^*) = \emptyset$

Step 2: Decision for termination: The process terminates here if $\mathcal{L} = \emptyset$ and (x^*, y^*) is the optimal solution for the problem, otherwise go to step 3.

Step 3: Node selection: Select a node, say N_i with corresponding LP relaxation LP_i, and remove it from the set $\mathcal{L}$

Step 4: LP relaxation: If LP_i is infeasible, then go back to step 3; otherwise, write the optimal solution to LP_i as (x^i, y^i) with the optimal value z_i

Step 5: Decision for pruning: in this step, we have three different cases:

(a) If $z_i \leq \underline{z}$, then it is pruned by bound and go back to step 3.
(b) If $z_i \geq \underline{z}$ and (x^i, y^i) is feasible to the original MILP problem, then update $\underline{z} = z_i$ and $(x^*, y^*) = (x^i, y^i)$ and then go back to step 3.
(c) If $z_i \geq \underline{z}$ but (x^i, y^i) is infeasible to the original MILP problem, then go to step 6

Step 6: Cut or Branch: For further exploration on the current node, we can do either of the following:

- Add a cutting plane to strength the LP_i and go back to step 4, OR
- Use the branch method to continue branch the current node to $k \geq 2$ sub-problems with LP relaxations $LP_{i_1}, \dots, LP_{i_k}$, as creating new sub-nodes as $N_{i_1}, \dots, N_{i_k}$ and add them to $\mathcal{L}$. Go back to step 1.

3.7 Problem Formulation

Bank Clearing Problem (BCP) is a discrete optimization problem that maximizes the total value of settled payments with the limited capacity for each participant, which can be formulated as a MILP problem with objective function as the value of settled payments and constraints that the total value sent by each bank cannot exceed its capacity. Variables of each payment are binary as 0 represents not settled payment and 1 represents settled payment.

To formulate the BCP as a MILP problem, let us clarify the notations as follows:

- n: the number of banks in the payment system
- b_i: the capacity of bank i
- m_i: the number of payments whose sender is bank i, $i = 1, \ldots, n$
- $r_{i,k}$: the receiver of the k-th payment in bank i's queue, $i = 1, \ldots, m_i$
- $v_{i,k}$: the value of the k-th payment in bank i's queue, $i = 1, \ldots, m_i$
- $x_{i,k} \in \{0, 1\}$: the decision variable of the k-th payment in bank i's queue, $i = 1, \ldots, m_i$, where 0 represents inactive payment and 1 represents active payment

Define $V(x)$ as the total value of the settled payments, $S_i(x)$ as the total value of the settled payments sent by bank i and $R_i(x)$ as the total value of the settled payments received by bank i, i.e.

$$\begin{aligned}
V(x) &= \sum_{i=1}^{n} \sum_{k=1}^{m_i} x_{i,k} v_{i,k} \\
S_i(x) &= \sum_{k=1}^{m_i} x_{i,k} v_{i,k} \quad i = 1, \ldots, n \\
R_i(x) &= \sum_{j=1}^{n} \sum_{k=1}^{m_j} x_{j,k} v_{j,k} \mathbb{1}_{\{r_{j,k}=i\}} \quad i = 1, \ldots, n
\end{aligned} \tag{3.7.1}$$

Then based on the definition of BCP, we can formulate the BCP as the follwing MILP problem:

$$\begin{aligned}
\text{maximize} \quad & V(x) \\
\text{subject to} \quad & S_i(x) - R_i(x) \le b_i, \qquad i = 1, \ldots, n, \\
& \qquad x_{i,k} \in \{0, 1\}, \quad k = 1, \ldots, m_i, \quad i = 1, \ldots, n
\end{aligned} \tag{3.7.2}$$

where V, S_i and R_i are functions defined above in (3.7.1)

3.8 Gurobi Solver

Gurobi solver is a solver for optimization problems that is interfaced with many programming languages such as Python, C++, R, etc and we use Gurobi solver in Python to solve the BCP problem. Figure 3.7 describes the process of Gurobi solver for a MILP problem.

Notice that the Gurobi solver does not guarantee the global optimum as it always returns the best optimal solution within a specific tolerance with default value 1e-4, i.e. solving to optimality. There is a difference between the concept of a global optimum and solving to optimality. For a global optimum, it is exactly the optimized value and for example for a maximization problem, if x^* is the global optimizer with the global optimum z^*, then mathematically it is equivalent to say that for any x in the feasible region with corresponding objective value z, we have $z \le z^*$. While for solving to optimality, usually there exists some parameter such as error or tolerance so that when some optimal value meets the specific requirement such as within the tolerance interval, the process will be stopped and this value will be taken as the optimum. However, it is possible that there exists some value that is more optimal than the current solution.

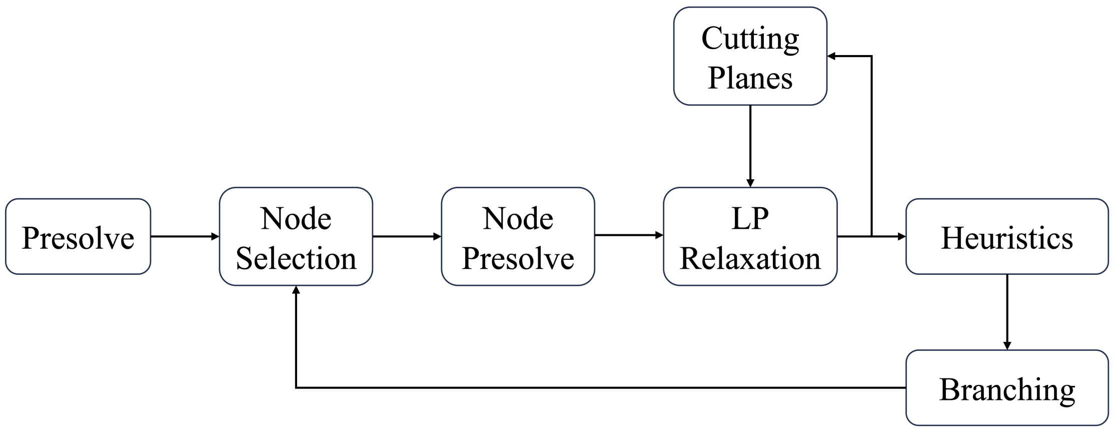


Figure 3.7: Procedure of Gurobi Solver for a MILP Problem

Chapter 4

Data

To evaluate the performances of the algorithms corresponding to queues with different characteristics in the interbank payment system, we first generate some artificial queues for different scenarios, using the Soramäki-Cook Algorithm [16] for the construction of the interbank-payment network and assigning the payment values following a truncated log-normal distribution with different standard deviation.

4.1 Network Construction

Note that the interbank payment system is actually a network, where each node represents a bank, the directed link from node i to node j represents a payment from bank i to bank j and the weight of the link equals the value of this payment. For the pseudo-code for the Soramäki Cook Algorithm see Algorithm 3 and relevant terminologies are explained as follows:

- n_0: initial number of nodes
- n: desired number of nodes (total number of banks)
- nb_payments: total number of payments in the queue
- $h = (h_i)_{i=1,\ldots,n}$: tracks the amount of preferential attachment strength of each bank, representing the relative possibility to be selected as a sender or receiver of a payment in the process of building payments network
- α: strength of preferential attachment, i.e. added to h_i when a payment is related to bank i
- matrix $s = (s_{ij})_{i,j=1,\ldots,n}$: s_{ij} represents the number of payments from bank i to bank j

4.2 Value Assignment

Assume that the values of payments in the queue follow a truncated log-normal distribution with a constant mean, a specific standard deviation, and also with an upper limit of 10 times the mean to cut off extremely large payments.

We generate the variates following the truncated log-normal distribution and randomly assign the number to each payment in the network as the value of the payment, i.e. the weight of each edge in the network.

Algorithm 3 Soramäki Cook Algorithm

```
Input  : $n_0$, n, nb_payments, $\alpha$
Output: data frame of one queue
Initialize $h[i] = 0$ for $i = 0, \dots, n-1$
Set $h[i] = 1$ for $i = 0, \dots, n_0 - 1$
Set s = np.zeros((n, n), dtype=int)
Set m = int(nb_payments / (n-n0))
for k in range($n_0$,n,1) do
    for l in range(0,m,1) do
        Randomly choose bank $i^*$ as the sender based on probability distribution $h$
        // i.e.bank $i$ has the probability $\frac{h_i}{\sum h_i}$ of being chosen
        Set $h[i^*] = h[i^*] + \alpha$
        Randomly choose bank $j^*$ as the receiver (Repeat until $j^* \neq i^*$)
        Set $h[j^*] = h[j^*] + \alpha$
        Set $s[i^*, j^*] = s[i^*, j^*] + 1$
    end
    Set $h[k] = 1$ // add a new node (bank)
end
for l in range(nb_payments-m*(n-$n_0$)) do
    Randomly choose bank $i^*$ as the sender based on probability distribution $h$
    Set $h[i^*] = h[i^*] + \alpha$
    Randomly choose bank $j^*$ as the receiver (Repeat until $j^* \neq i^*$)
    Set $h[j^*] = h[j^*] + \alpha$
    Set $s[i^*, j^*] = s[i^*, j^*] + 1$
end
```

Proposition 4.2.1 (Relationship between log-normal distribution and normal distribution)**.** *Let X be a normal distribution with mean μ_x and variance σ_x^2 and $Y = e^X$ be the corresponding log-normal distribution with mean μ_y and variance σ_y^2. Then:*

$$\mu_x = \ln \frac{\mu_y}{\sqrt{\frac{\sigma_y^2}{\mu_y^2} + 1}} \qquad \sigma_x^2 = \ln\left(\frac{\sigma_y^2}{\mu_y^2} + 1\right) \tag{4.2.1}$$

Proof. Note that the moment generating function of normal distribution X is:

$$E[e^{tX}] = M_X(t) = e^{\mu_x t} e^{\frac{1}{2}\sigma_x^2 t^2}$$

Then by the fact that $Y = e^X$, we have:

$$\begin{aligned} \mu_y &= E[Y] = E[e^X] = M_X(1) = e^{\mu_x + \frac{\sigma_x^2}{2}} \\ \sigma_y^2 &= E[Y^2] - E[Y]^2 = M_X(2) - \mu_y^2 = e^{2\mu_x + 2\sigma_x^2} - e^{2\mu_x + \sigma_x^2} = \mu_y^2(e^{\sigma_x^2} - 1) \end{aligned}$$

Solving the system of two equations, we can get the results (4.2.1) □

We use *scipy.stats.truncnorm* in Python to generate a truncated normal variate, where the mean and standard deviation are computed by the relationship in proposition 4.2.1 and the upper bound is set to be $\ln(10\mu_y)$. Then exponentiating the variate gives an approximate generation of a truncated log-normal variate. Table 4.1 lists the statistics of the distribution of 100,000 samplings and the significantly small percentage errors show that this generation algorithm approximately generates the truncated log-normal distribution with supposed mean and proportional sigma. Figure 4.1 illustrates the histogram of 100,000 samplings.

Combining the value of each payment with the corresponding network of queuing payments generated in section 4.1, we simulate one queue as Figure 4.2 shows.

supposed std/mean	realized mean	realized std	max	realized std / mean	(%) mean error	(%) std error
0.05	1937127.46	96904.46	2373619.50	0.05	-0.00	0.05
0.10	1937436.24	192752.33	2920497.62	0.10	0.02	-0.50
0.20	1933736.40	385152.53	4129834.47	0.20	-0.18	-0.59
0.30	1936961.48	580539.74	6990150.51	0.30	-0.01	-0.10
0.40	1944428.28	784768.25	9735584.29	0.40	0.38	1.28
0.50	1932845.32	968036.82	12941281.18	0.50	-0.22	-0.05
0.60	1939360.60	1160781.27	17285879.75	0.60	0.12	-0.13
0.70	1933540.48	1341037.07	19298776.59	0.69	-0.19	-1.10
0.80	1931878.74	1524915.89	19158582.56	0.79	-0.27	-1.60

Table 4.1: Generating Truncated Log-normal Distribution under Different Coefficients of Variation

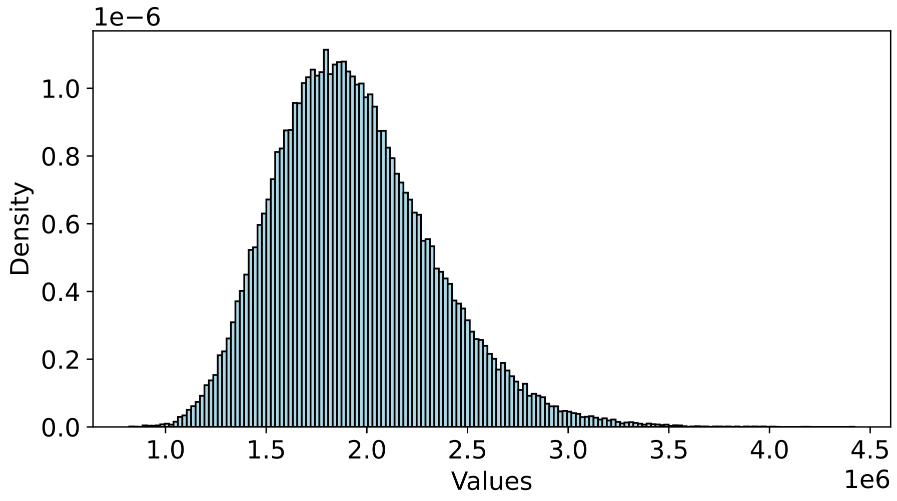


Figure 4.1: Histogram of 100,000 Samples from Truncated Log-normal Distribution

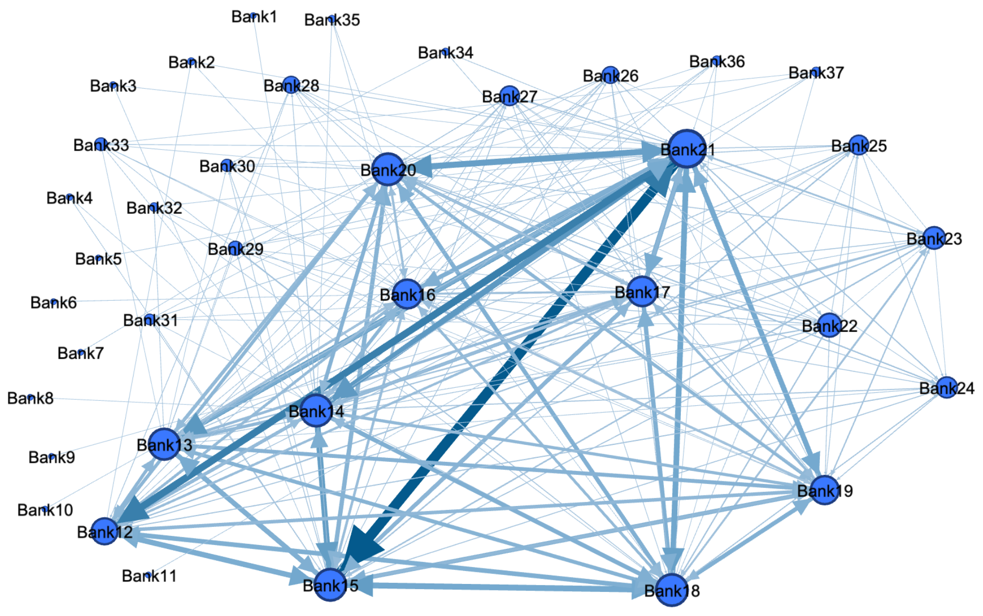


Figure 4.2: Network of One Queue of Payments

Chapter 5

Results

In this chapter, we simulate 51 scenarios of queues by changing different values of four parameters and simulating 1,000 queues for each scenario. Then we evaluate and compare the performances of the solutions given by different algorithms, using error bars to show the 99% confidence interval of the mean.

For the implementation of the BM Algorithm, we solve the MILP problem by running the Gurobi solver in Python, version 10.0.1. The configuration of the computer is MacBook Pro 2021 with an Apple M1 Pro chip, 10 cores and 32GB Unified Memory. Note that the computing time depends on the computer configuration.

5.1 Parameters

During the simulations, we fix the following parameters:

- $n_0 = 10$: the number of initial nodes for the construction of the queue by Algorithm 3
- $n = 37$: the total number of banks in the interbank payment system
- mean = 98.6e12/50.9e6: the average value of one payment in the queue, settled as a constant mean for the truncated log-normal distribution for values of the payments (this statistical data based on the annual data provided by the Bank of England[1])
- upper bound = 10 $\times$ mean: pre-assigned upper bound of the truncated log-normal distribution of all payments in one queue for value assignment

Then we do the simulations on different values on the following parameters to simulate different performances of each algorithm under queues with different features:

- Coefficient of Variation (CV): we set the standard deviation of the truncated log-normal distribution of values of payments in the queue as a CV with respect to mean, i.e. larger CV implies larger volatility for the value in a queue.
- Number of Payments: the number of payments in one queue represents the size of the queue
- Capacity: we set capacity as the proportion of the total liquidity needed to settle all the payments in the queue, i.e. for each bank, its liquidity needed is the difference between the sum of its send-out payments value and the sum of its receive-in payments value, and smaller capacity represents more serious lack of liquidity in the market.
- α: the strength of preferential attachment, i.e. larger α implies that more payments will be related to several big banks (i.e. core banks) while fewer payments will involve other banks (i.e. periphery banks).

[1]https://www.bankofengland.co.uk/payment-and-settlement/chaps

Parameter	Value	Total
CV	0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8	9
Number of Payments	100, 200,300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000, 50000, 100000	22
Capacity	10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%	10
α	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0	10

Table 5.1: Scenarios of Simulations

Table 5.1 lists all the scenarios we set for the four parameters. And Table 5.2 lists the default parameters when simulating each parameter.

Parameter	Default Value
CV	0.2
Number of Payments	3000
Capacity	80%
α	0.1

Table 5.2: Default Parameters for Simulations

5.2 Outputs

For each queue, we implement 4 Algorithms and denote them in the plot legends as follows:

- Bech Soramäki 1: Algorithm 2 (Bech-Soramäki Algorithm) with descending sorting of payment values, i.e. large payment settled first
- Bech Soramäki 2: Algorithm 2 (Bech-Soramäki Algorithm) with ascending sorting of payment values, i.e. small payment settled first
- Güntzer et al. 5: Algorithm 1 (Güntzer5 Algorithm)
- MILP: BM Algorithm

We optimize the maximum settled value and compute the following data to analyze the performances of each algorithm. Figure 5.1 and Figure 5.2 give two different layouts of example outputs for 5 queues by Güntzer5/Bech-Soramäki Algorithm and BM Algorithm, respectively:

- value settled: the total value of settled payments
- total value: the total value of payments in the queue
- % value settled: the percentage of value settled with respect to the total value, i.e.

$$\% \text{ value settled} = \text{value settled} \ / \ \text{total value} \times 100\%$$

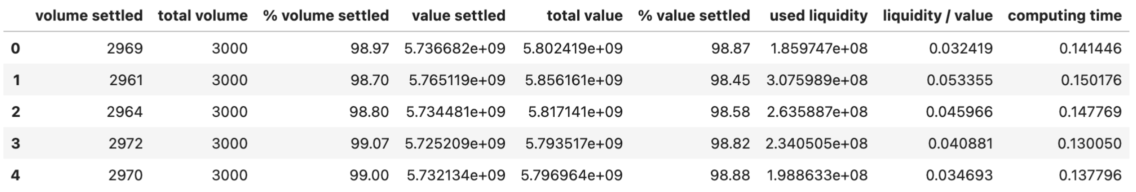
- volume settled: the total number of settled payments
- total volume: the total number of payments in the queue
- % volume settled: the percentage of volume settled with respect to the total volume, i.e.

$$\% \text{ volume settled} = \text{volume settled} \ / \ \text{total volume} \times 100\%$$

- used liquidity: the total liquidity used for settling these payments, defined as:

$$\text{used liquidity} = \textstyle\sum_{i=1}^{n} \max\{\text{inital balance of bank } i \text{ - final balance of bank } i,\, 0\}$$

- liquidity / value: relative used liquidity with respect to the value settled.
- computing time: time (in seconds) for solving the problem for one queue by each algorithm

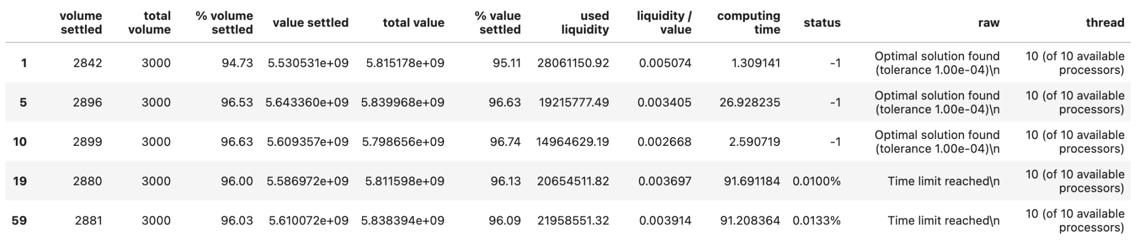


	volume settled	total volume	% volume settled	value settled	total value	% value settled	used liquidity	liquidity / value	computing time
0	2969	3000	98.97	5.736682e+09	5.802419e+09	98.87	1.859747e+08	0.032419	0.141446
1	2961	3000	98.70	5.765119e+09	5.856161e+09	98.45	3.075989e+08	0.053355	0.150176
2	2964	3000	98.80	5.734481e+09	5.817141e+09	98.58	2.635887e+08	0.045966	0.147769
3	2972	3000	99.07	5.725209e+09	5.793517e+09	98.82	2.340505e+08	0.040881	0.130050
4	2970	3000	99.00	5.732134e+09	5.796964e+09	98.88	1.988633e+08	0.034693	0.137796

Figure 5.1: Example Outputs for 5 Queues by Güntzer5/Bech-Soramäki Algorithm

For our BM Algorithm, we additionally have three more columns as follows:

- status: -1 represents the success of finding an optimal solution; if an optimal solution cannot be found in the settled limit time (90s), then the gap between the current solution and the upper bound will be shown in this column
- raw: the raw column shows the optimality of the solution given by Gurobi within a limited processing time. If an optimal solution is found then it shows "Optimal solution found" with corresponding tolerance; if the optimal solution is not found within the limited time (we set 90 seconds in all simulations), then it marks as "Time limit reach"
- thread: the thread column collects the information of the number of threads used for solving the problem as more threads are used, the much faster the solution will be reached



	volume settled	total volume	% volume settled	value settled	total value	% value settled	used liquidity	liquidity / value	computing time	status	raw	thread
1	2842	3000	94.73	5.530531e+09	5.815178e+09	95.11	28061150.92	0.005074	1.309141	-1	Optimal solution found (tolerance 1.00e-04)\n	10 (of 10 available processors)
5	2896	3000	96.53	5.643360e+09	5.839968e+09	96.63	19215777.49	0.003405	26.928235	-1	Optimal solution found (tolerance 1.00e-04)\n	10 (of 10 available processors)
10	2899	3000	96.63	5.609357e+09	5.798656e+09	96.74	14964629.19	0.002668	2.590719	-1	Optimal solution found (tolerance 1.00e-04)\n	10 (of 10 available processors)
19	2880	3000	96.00	5.586972e+09	5.811598e+09	96.13	20654511.82	0.003697	91.691184	0.0100%	Time limit reached\n	10 (of 10 available processors)
59	2881	3000	96.03	5.610072e+09	5.838394e+09	96.09	21958551.32	0.003914	91.208364	0.0133%	Time limit reached\n	10 (of 10 available processors)

Figure 5.2: Example Outputs for 5 Queues by BM Algorithm

For each parameter variable, we evaluate and compare the four main statistics of optimal solutions as follows:

- % volume settled: the percentage of volume settled over the total volume of the queue
- % value settled: the percentage of value settled over the total value of the queue
- efficiency: value / liquidity, i.e. The value settled that one unit of liquidity used
- computing time: the computing time for each algorithm to compute the optimal solution for each queue.

We take the confidence interval for each statistic computed by 1000 simulations as a main method to evaluate and compare the performance of each algorithm. Table 5.3 gives an example of the computed statistics with 99% confidence interval by default parameters listed in Table 5.2.

	Bech-Soramäki 1	Bech-Soramäki 2	Güntzer5	MILP
% volume settled	96.078±0.0933	97.8817±0.0512	98.9888±0.0116	98.7138±0.017
% value settled	97.3099±0.0659	96.8644±0.0742	98.8243±0.0147	98.8594±0.0147
efficiency	29.0642±0.4946	30.1948±0.538	29.0141±0.4831	28.8082±0.4767
computing time	0.0955±0.0006	0.0838±0.0004	0.1469±0.0016	1.0153±0.0052

Table 5.3: Confidence Intervals for Outputs

5.3 Simulations

We use the error-bar plot to illustrate the confidence interval we compare our BM Algorithm with the other three algorithms based on the four error-bar plots of four statistics listed in section 5.2. Furthermore, additional graphs and tables are included for further analysis.

In all simulations, our BM Algorithm has the best performance on average on the percentage of value settled. In most scenarios, Güntzer5 Algorithm performs best on the percentage of volume settled. Moreover, we set the time limit for computing as 90 seconds. There are 24 queues that computed to the time limit and the current best solution is returned. However, all these sub-optimal solutions still outperform the other three algorithms on the percentage of settled payment values.

5.3.1 Simulations on Coefficient of Variation (CV)

In this section, we simulate different volatilities of the values of payments in the queue by taking different coefficients of variation (CV) with respect to the mean, from 0.1 to 1.0 with step 0.1. Other parameters are set to default as Table 5.2. Since we set the mean as a constant, when CV increases, the volatility of the payment value increases.

According to Figure 5.3, the change of the features of relative size between each payment does not have some significant impact on the percentages of volume and value settled for both BM Algorithm and Güntzer5 Algorithm and Güntzer5 Algorithm always performs better for volume while relatively worse for value than BM Algorithm. Both of them outperform the Bech-Soramäki Algorithm with ascending or descending order significantly for value, but when CV is larger than 0.4, Bech-Soramäki Algorithm with ascending order, i.e. small payments settled first, has better performance on volume than BM Algorithm.

The efficiency of all algorithms decreases when CV increases, as when the volatility of payment values becomes larger, one unit liquidity settles less value, and Bech-Soramäki Algorithm with ascending order has the largest efficiency much more than the other three however, it is worth noticing that Bech-Soramäki Algorithm settled the least payments.

The computing time for our BM Algorithm is relatively larger than the other three algorithms, but it is still stable for around one second which is already very fast and does not oscillate as CV

5.3.2 Simulations on Number of Payments

In practice, different systems may have different sizes of participating banks, or usually more payments are submitted at the beginning of the day, either of which results in different numbers of payments in the queue. In this section, we range the number of payments from 100 to 100,000 to compare the performance of each algorithm under different sizes of the queue. Other parameters are set to default as Table 5.2. Since the range of parameters goes from 100 to 100,000, we use the log scale of the x-axis to better illustrate the shape of the curve.

As Figure 5.4 shows, as the number of payments increases, both the percentages of volume settled and value settled increase and almost converge starting from 3000 payments in a queue. But notice that Güntzer5 Algorithm does not perform best for volume when considering small queues with sizes less than 1,000. Both of them outperform a lot over the Bech-Soramäki Algorithm for both descending and ascending order of the payment values in both percentages of volume settled and value settled. As the curve of the efficiency for each algorithm is almost overlapping, the efficiency is almost the same and increases as the number of payments increases. Also as Figure 5.5 shows, the computing time for all algorithms increases linearly with respect to the number of payments, which makes it easy to estimate the computing time for each algorithm when the number of payments in the queue is changed.

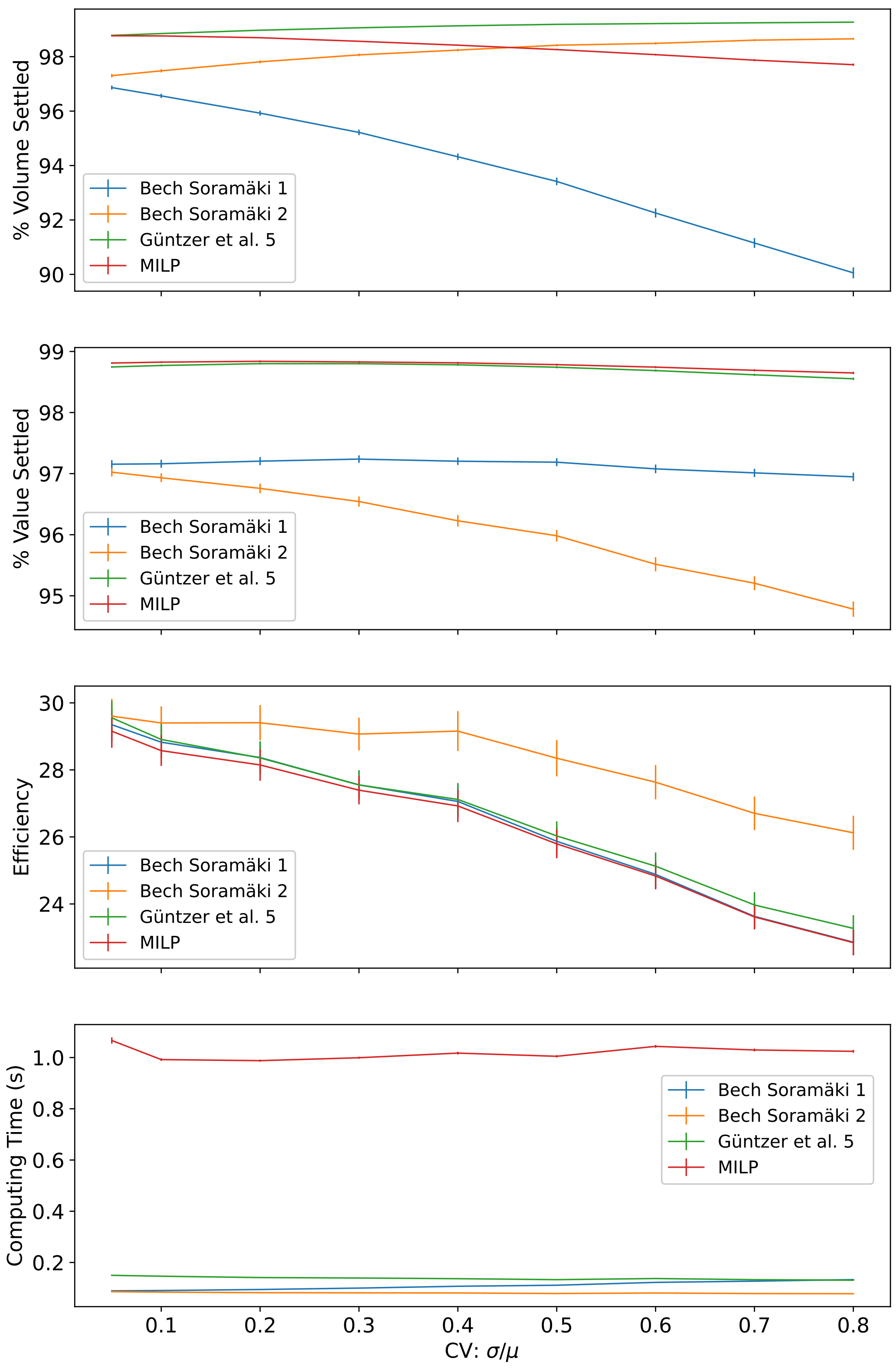


Figure 5.3: Performances for Different CVs (σ/μ)

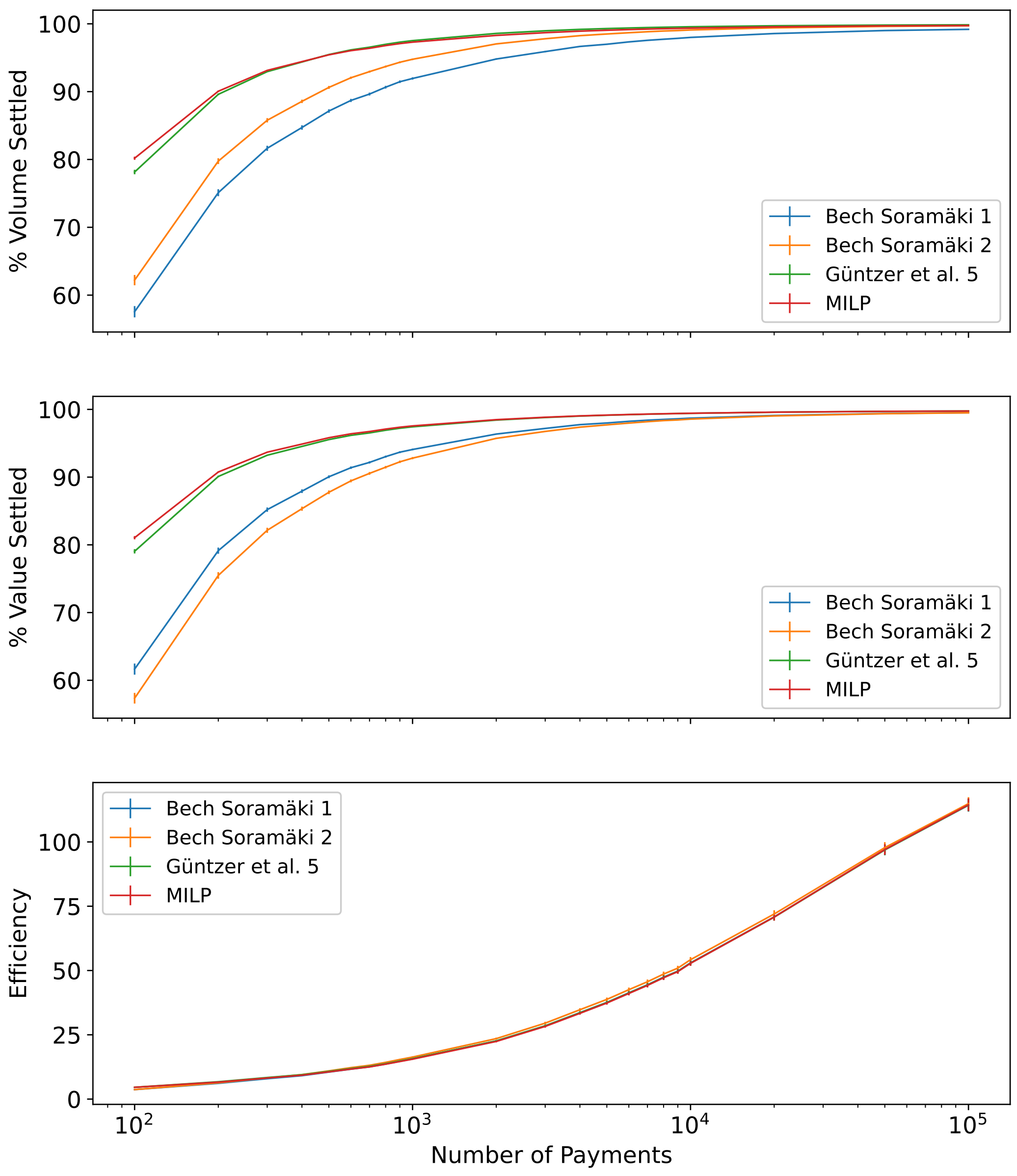


Figure 5.4: Performances for Different Numbers of Payments

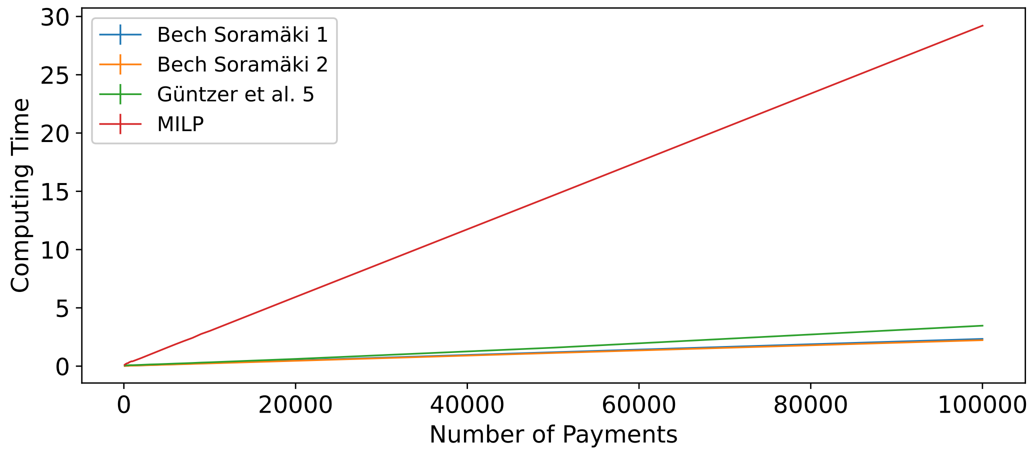


Figure 5.5: Computing Time for Different Numbers of Payments

5.3.3 Simulations on Capacity

In this section, we simulate different situations of the relative shortage of liquidity in the market, i.e. larger capacity represents more current liquidity that can be used to settle the payments. We take different capacities from 10% to 100% with step 10% to compare the performances and note that when capacity = 100%, all payments can be settled without any netting. Other parameters are set to default as Table 5.2.

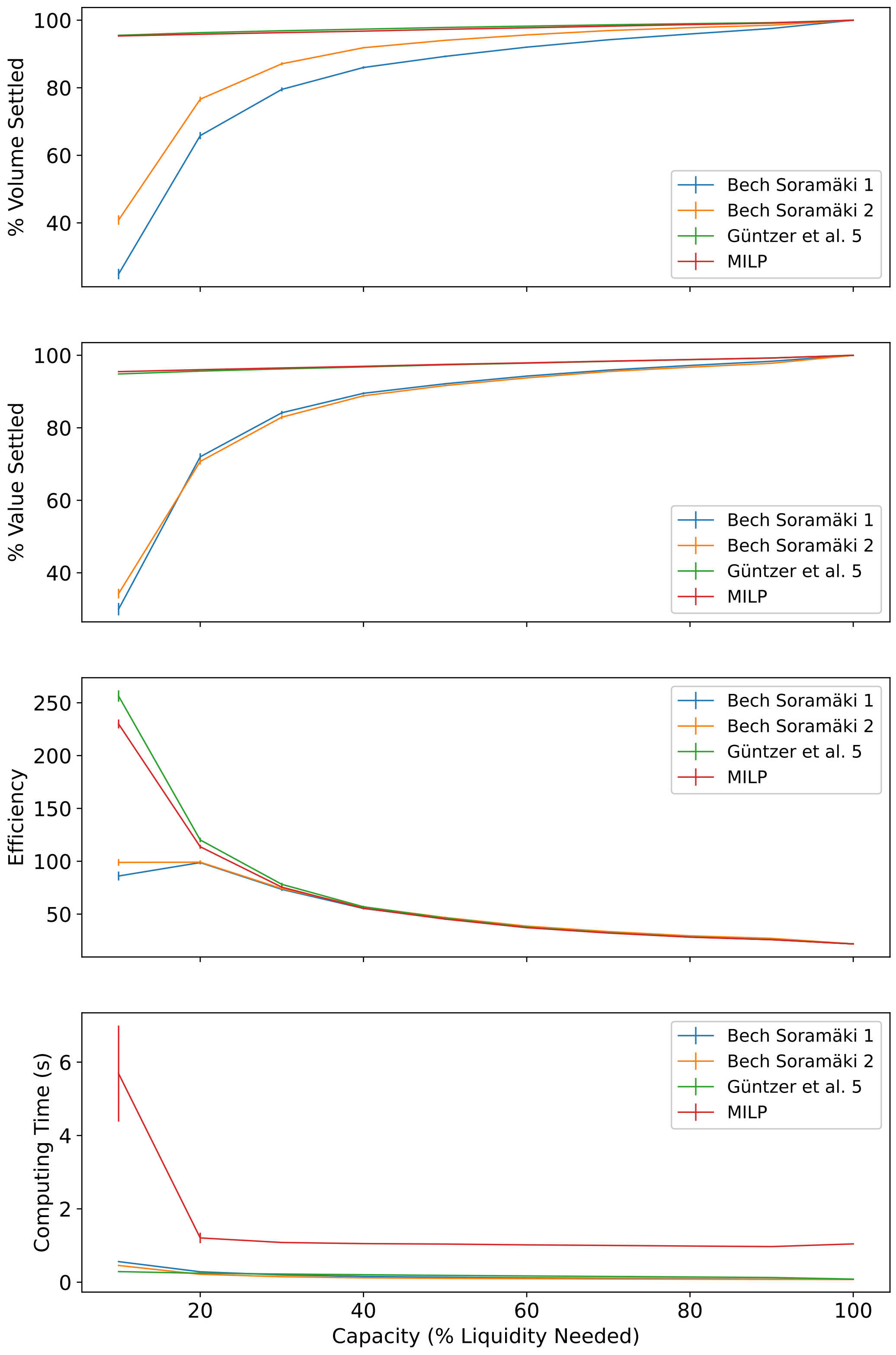


Figure 5.6: Performances for Different Capacities (% liquidity needed)

Under different capacities, Figure 5.6 shows that BM Algorithm and Gunzter5 Algorithm perform well for all situations of existing liquidity in the market as they can always settle more than 95% volume and value of the payments and the performance difference between these two algorithms is not significant. However, for Bech-Soramäki Algorithm, the amount of current liquidity in the market has a significant influence on the algorithm performance, especially when there is a serious lack of liquidity in the market. Similar results are also given as Güntzer5 and BM Algorithms always outperform the Bech-Soramäki Algorithm.

Based on the stable performance on volume and value for Güntzer5 Algorithm and BM Algorithm, their efficiency decreases as the capacity increases, significantly for a small increase when the current liquidity is very low, while for Bech-Soramäki Algorithm, the efficiency also decreases but without obvious change. And when there is more than 30% of liquidity needed available in the market, the difference among the efficiencies of all algorithms is extremely slight.

Analyzing the computing time for simulations on capacity, we can take 0.1 capacity as a special case because, during the 1,000 simulations, there exist 24 queues that BM Algorithm does not return the optimal solution since the time limit of 90 seconds is reached and the Gurobi solver provides only the current solution (not optimal). Therefore, the average and the confidence interval of computing time when the capacity is 0.1 are relatively large. For all other capacities, BM Algorithm uses only about 1 second on average to compute the optimal solution for a queue with 3,000 payments, which is fast enough although the computing time is relatively larger than that of the other three algorithms.

For capacity, Figure 5.7 shows the difference between the total value of settled payment between the BM Algorithm and Güntzer5 Algorithm ($V_M - V_G$) for different capacities. It is obvious that the difference decreases exponentially as the capacity increases, which implies that when there is a serious lack of liquidity in the market, then the payment system with the BM Algorithm settles more value of payments than Güntzer5 Algorithm and the serious lack, the better performance that BM Algorithm can give than Güntzer Algorithm. This result strongly encourages us to use our BM Algorithm as the computing time for these two algorithms does not differ too much, i.e. around 1 second for most capacities and only 5 seconds for 10% capacity which is due to some Time Limit Reached cases for some queues during the simulations.

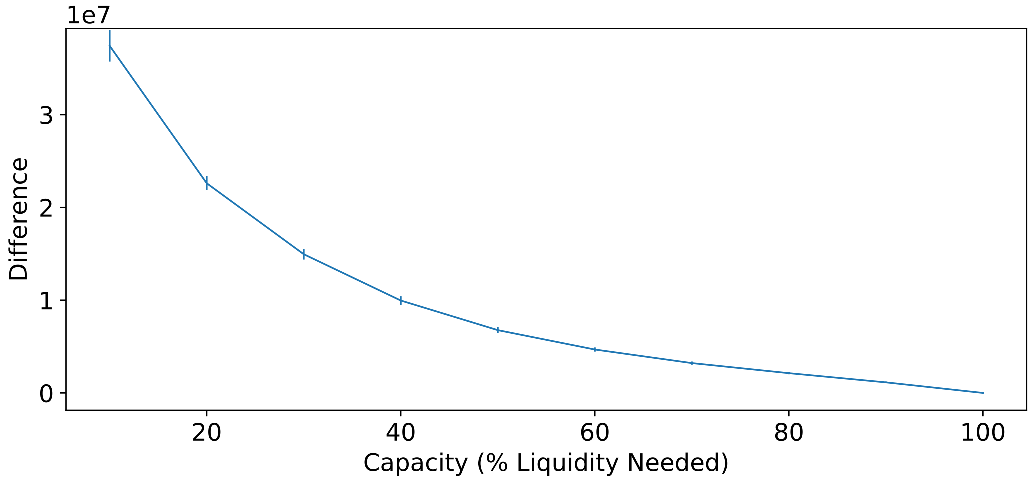


Figure 5.7: Different Total Value of Settled Payments between BM Algorithm and Güntzer5 Algorithm for Different Capacities

5.3.4 Simulations on α

In this section, we take different α ranging from 0.1 to 1.0 with step 0.1 to simulate different features of the network construction of the queue. Based on Soramäki Algorithm, α represents the strength of preferential attachment, i.e. during the network construction, any time a bank (node) is selected to be a sender or a receiver of the payment (linked with the edge), the probability for this bank being selected next time is increased by adding α to the probability scale.

Figure 5.8 illustrates the performances under different α. First of all, Güntzer5 Algorithm performs always the best for the percentage of volume for all α's with the smallest 99% confidence interval, while for the average percentage of value settled in 1000 simulations, BM Algorithm performs the best. Both of Güntzer5 and BM Algorithms outperform a lot over the Bech-Soramäki Algorithm for both descending and ascending order of the payment values.

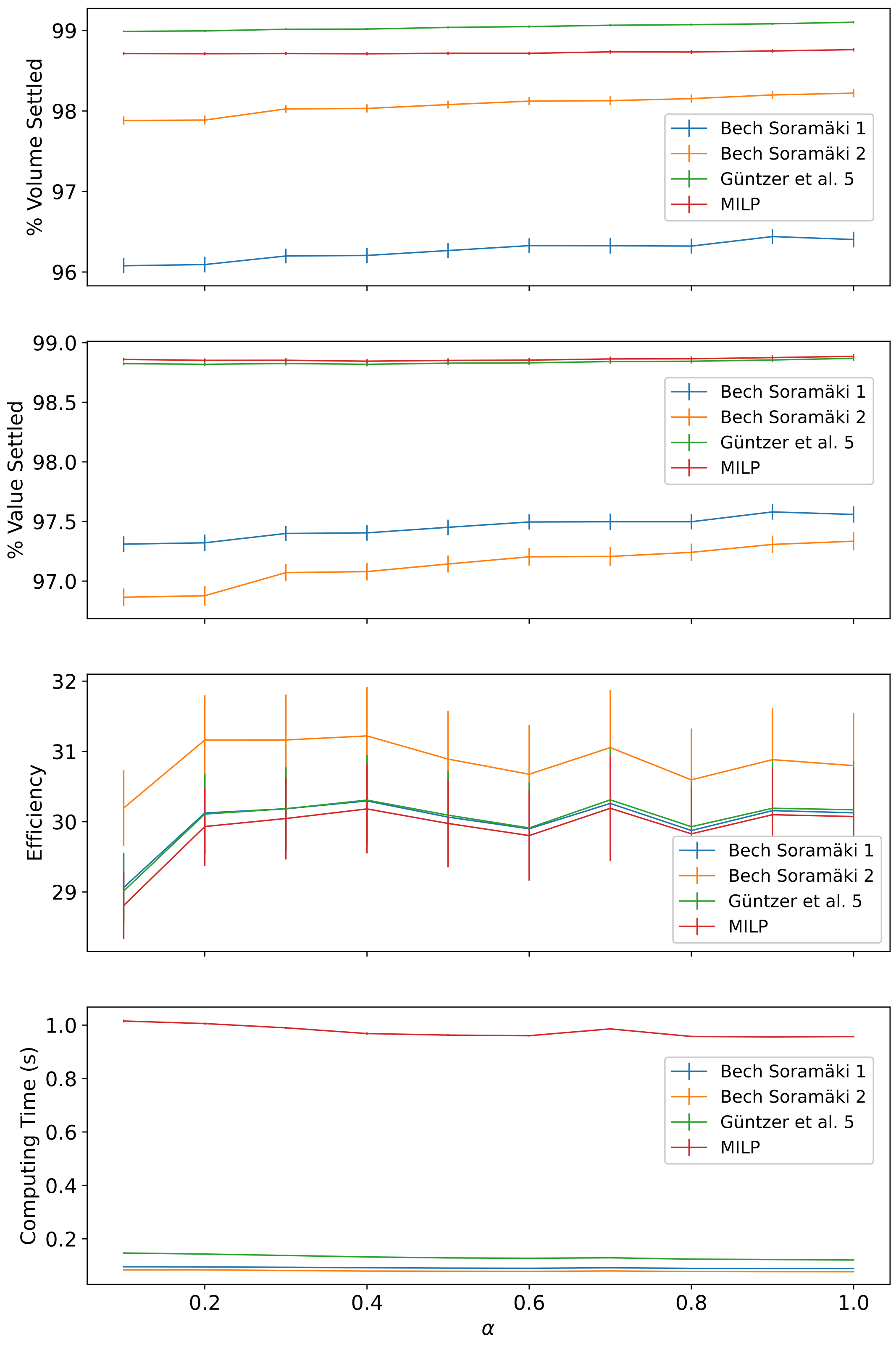


Figure 5.8: Performances for Different α

Chapter 6

Further Discussion

6.1 Change of Objective Function

Back to the motivation of our BM Algorithm, we can simply change the optimization target by changing the objective function in the MILP problem and one of the important applications is to maximize the delay time for settled payments, i.e. minimize the delay time for remaining payments in the queue.

The delay time for payment is the time difference between the current time and the original time that the payment entered the system, i.e. the time it has been waited to be settled. To further discuss the optimization problem of delay time, we assume that the delay time for each payment in the queue follows a truncated log-normal distribution with an average time of 10 minutes, a standard deviation of 5 minutes, and an upper bound of 60 minutes, i.e. any payment delayed for more than 1 hour will be rejected and removed from the queue.

Let $d_{i,k}$ be the delay time of the k-th payment in bank i's queue. To be consistent with the notation in section 3.7, define the total delay time $D(x)$ as:

$$D(x) = \sum_{i=1}^{n} \sum_{k=1}^{m_i} x_{i,k} d_{i,k} \tag{6.1.1}$$

Substituting $D(x)$ for $V(x)$ in MILP version of BCP (3.7.2), we have:

$$\begin{aligned} &\text{maximize} && D(x) \\ &\text{subject to} && S_i(x) - R_i(x) \leq b_i, \qquad i = 1, \ldots, n, \\ &&& x_{i,k} \in \{0,1\}, \quad k = 1, \ldots, m_i, \quad i = 1, \ldots, n \end{aligned} \tag{6.1.2}$$

where S_i and R_i are the total value of send-out payments and receive-in payments of bank i, respectively, defined in (3.7.1)

6.2 Multi-Objective Functions

To further discuss another motivation of our BM Algorithm that we can have multiple optimization targets with different priorities, we consider two cases as setting the maximization of the total value of settled payments to be the first objective function and then making the second objective function to be the maximization of the total volume in section 6.2.2 or the total delayed time of settled payments in section 6.2.3, respectively.

Also, in this section, we compare the results given by multi-objective algorithms with the results of the original single-objective BM Algorithm. For the additional objective function with respect to volume, we also compare the performance with Güntzer5 Algorithm since Güntzer5 Algorithm always outperforms others in the total volume of settled payments. And for the additional objective function with respect to delay time, we also compare the performance with Bech-Soramäki Algorithm since Bech-Soramäki Algorithm can take the delay time order into consideration by sorting.

6.2.1 Multi-Objective Problem

We choose to investigate multi-objective optimization using a hierarchical approach. This approach assigns a priority to each optimization function and a relative tolerance. The solver then processes the optimization functions by priority order and allows the later objectives to degrade the previous ones within their pre-assigned relative tolerance. For more details on the method, see [17]. To exemplify, consider a duo-objective MILP problem with objective functions $f(x)$ in higher priority and $g(x)$ in lower priority and a feasible region D. Let r be the relative tolerance. Solving the first MILP problem:

$$\max\{f(x) : x \in D\}$$

Suppose the optimal solution is x^* with optimal value z^*. Then it solves the optimization problem for the second objective function $g(x)$ as follows:

$$\max\{g(x) : x \in D \quad \text{and} \quad f(x) \geq z^* - |z^*| \times r\}$$

where a degrading bound restriction is added for the first objective function related to the pre-assigned relative tolerance.

6.2.2 Value-Volume

Consistent with the notation in section 3.7, since $x_{i,k}$ is a binary variable for each payment, we can calculate the volume of settled payments $Vol(x)$ by summing all variables $x_{i,k}$ as:

$$Vol(x) = \sum_{i=1}^{n}\sum_{k=1}^{m_i} x_{i,k} \tag{6.2.1}$$

Then the MILP problem with muti-objective functions is defined as follows:

$$\begin{aligned} \text{maximize} \quad & V(x) - 1st; Vol(x) - 2nd \\ \text{subject to} \quad & S_i(x) - R_i(x) \leq b_i, \qquad i = 1, \ldots, n, \\ & \qquad x_{i,k} \in \{0,1\}, \quad k = 1, \ldots, m_i, \quad i = 1, \ldots, n \end{aligned} \tag{6.2.2}$$

According to the simulation results in chapter 5, it is worth noticing that the Güntzer5 Algorithm performs best in the volume of settled payments in most scenarios. Therefore, we compare the result of the duo-objective with value first and volume second with the result of the Güntzer5 Algorithm and the single-objective BM Algorithm.

Table 6.1 shows the statistics of the performances for three algorithms under the parameters that CV = 0.2, number of payments in the queue = 3000, capacity = 80% and $\alpha = 0.1$. Notice that compared with the optimal solution to the single-objective MILP problem that optimizes the total value settled, MILP-Multi settles about 9 more payments on average of 1000 simulation queues at the cost of only 0.15% value of payments. Moreover, MILP-Multi also settles 1 more payment than Güntzer5 Algorithm on average and has the largest efficiency among the three algorithms.

	Güntzer5	MILP	MILP-Multi
volume settled	2969.326±0.3584	2961.145±0.5337	2970.509±0.344
value settled (%)	98.8117±0.0153	98.8482±0.0153	98.6929±0.0162
efficiency	28.8148±0.5077	28.5983±0.4993	29.7576±0.5402
computing time	0.1466±0.0008	1.0304±0.0038	0.9167±0.0152

Table 6.1: Results Comparison among Algorithms for Multi-Objective with Value and Volume

6.2.3 Value-Delay

Then the MILP problem with muti-objective functions is defined as follows:

$$\begin{aligned} \text{maximize} \quad & V(x) - 1st; D(x) - 2nd \\ \text{subject to} \quad & S_i(x) - R_i(x) \leq b_i, \qquad i = 1, \ldots, n, \\ & \qquad x_{i,k} \in \{0,1\}, \quad k = 1, \ldots, m_i, \quad i = 1, \ldots, n \end{aligned} \tag{6.2.3}$$

Under the same assumption of delay time as in section 6.2.1, we simulate the delay time of payments in the queue as a truncated log-normal distribution with an average of 10 minutes, standard deviation of 5 minutes and upper bound of 60 minutes.

According to the introduction of Bech-Soramäki Algorithm in section 2.1.2, we can define the sequence constraint in GRP by sorting the payments in the queue descendingly in terms of the delay time so that payments with longer delay time will be settled first, which has the same purpose of maximizing the total delay time of settled payments (equivalently minimizing the delay time of remaining payments in the queue after then). Therefore, in this section, we compare the result of the duo-objective with value first and delay time second with the result of the Bech-Soramäki Algorithm and the single-objective BM Algorithm.

Table 6.2 shows the statistics of the performances for three algorithms under the parameters that CV = 0.2, number of payments in the queue = 3000, capacity = 80% and $\alpha = 0.1$. Notice that MILP-Multi gives an optimization solution with about 14 thousand seconds less delay time than that of the single-objective BM Algorithm at the cost of nearly 0.13% value of settled payments, and about 10 thousand seconds less than that of the Bech-Soramäki Algorithm with even more percentage of settled payments value. Moreover, MILP-Multi also settles the largest volume of payments and has the second-largest efficiency among the three algorithms, only 0.3 smaller than the Bech-Soramäki Algorithm on average of 1000 simulation queues.

	Bech-Soramäki	MILP	MILP-Multi
volume settled	2910.428±2.2188	2961.305±0.4981	2965.668±0.4002
value settled (%)	97.0065±0.0744	98.8512±0.0144	98.7284±0.0151
remaining delay (s)	20134.2371±524.6301	23290.3592±331.3129	9260.8113±125.3079
remaining delay (%)	1.1186±0.0292	1.2939±0.0184	0.5145±0.0069
efficiency	29.2497±0.5322	28.5613±0.4994	28.9882±0.5175
computing time	0.0886±0.0005	0.9812±0.0038	0.9798±0.0099

Table 6.2: Results Comparison among Algorithms for Multi-Objective with Value and Delay

Chapter 7

Conclusion

As more and more interbank payment systems use Liquidity Saving Mechanisms (LSM) to reduce liquidity needs and liquidity risk, it is important to find the best possible solution to the Bank Clearing Problem (BCP).

In this thesis, we propose a new approach, the BM Algorithm, to mathematically solve the BCP. The main idea is to explicitly formulate the BCP into a 0-1 MILP problem. Then we use the Branch-and-Cut method to get an optimal solution within the error tolerance of 1e-4 (using the Gurobi solver in Python). We compare our method to two well-known algorithms in the LSM literature: Guntzer5 and Bech-Soramäki. Our method allows for more flexibility in the objective function, including the possibility of multi-objective optimization.

Simulation results under different scenarios show that our new approach outperforms on average the two popular algorithms on the total value settled.

Although the Güntzer5 Algorithm performs best on the percentage of volume settled in most scenarios, our BM Algorithm does not have too much difference on the settled volume and does outperform the Güntzer5 Algorithm on the settled value.

In terms of computing time, it is worth noting that although our BM Algorithm runs slower than the other two algorithms, it is fast enough to be used in practice. For the default number of payments (3000), it just needs 1 additional second. Also, the computing time of the BM Algorithm increases only linearly with respect to the number of payments, which guarantees that even for the extremely large size of interbank payment systems, our algorithm will not take too much time to produce a solution.

Additionally, the brief discussion about the change of objective function shows that our BM Algorithm is much more flexible than the previous algorithms so that in practice, the system operator can optimize different targets, e.g. value, volume, delay time, etc., based on different requirements. The result of our BM Algorithm can also be used as a benchmark for other systems to compare with their algorithms.

One limitation we met is that the Gurobi solver sometimes gives different solutions for the same MILP problem because of the different setting order of the parameters, variables, equations, etc. The main limitation comes from the random selection of the nodes in the Branch-and-Cut method(see [18] for details). Further research that solves the BCP-MILP problem by other solvers should be done.

Secondly, we have already briefly discussed the MILP problem with duo-objective functions by the hierarchical approach. Further research on multi-objective problems should also be done, such as investigating more combinations of objective functions (e.g. value, volume and delay time) and quantifying the trade-offs between objective functions.

Finally, similar to the idea from the Bech-Soramäki Algorithm that introduces the sequence constraint to the BCP, further exploration could be done to take into consideration some additional order requirements for the payments in the queue, e.g. participant priorities.

Bibliography

[1] Morten L. Bech, Christine Preisig, and Kimmo Soramaki. Global trends in large-value payments. *SSRN Electronic Journal*, 14, 2008.

[2] Morten L. Bech and Bart Hobijn. Technology diffusion within central banking: The case of real-time gross settlement. *SSRN Electronic Journal*, 2006.

[3] Bank for International Settlements. Real time gross settlement systems prepared by the committee on payment and settlement systems of the central banks of the group of ten countries, basle. 03 1997.

[4] Morten Bech and Kimmo Soramäki. Liquidity, gridlocks and bank failures in large value payment systems. *E-Money and Payment System Review*, pages 113–127, 01 2002.

[5] Antoine Martin and James McAndrews. An economic analysis of liquidity-saving mechanisms. *SSRN Electronic Journal*, 2008.

[6] Michael M. Güntzer, Dieter Jungnickel, and Matthias Leclerc. Efficient algorithms for the clearing of interbank payments. *European Journal of Operational Research*, 106:212–219, 04 1998.

[7] Morten L. Bech and Kimmo Soramäki. Gridlock resolution in interbank payment systems. *Bank of Finland Research Discussion Paper*, 9, 06 2001.

[8] Yakov M Shafransky and Alexander Doudkin. An optimization algorithm for the clearing of interbank payments. *European Journal of Operational Research*, 171:743–749, 06 2006.

[9] Michele Conforti, Gẽrard Cornuẽjols, and Giacomo Zambelli. *Integer Programming.* Cham Springer International Publishing, 2014.

[10] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123:397–446, 11 2002.

[11] Egon Balas. Facets of the knapsack polytope. *Management Science Research Rept*, 8:146–164, 12 1975.

[12] Ralph Gomory. *Outline of an Algorithm for Integer Solutions to Linear Programs and An Algorithm for the Mixed Integer Problem*, pages 77–103. 01 2010.

[13] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 04 1973.

[14] Hanif D. Sherali and Warren P. Adams. A hierarchy of relaxations and convex hull characterizations for mixed-integer zero—one programming problems. *Discrete Applied Mathematics*, 52:83–106, 07 1994.

[15] Franklin Djeumou Fomeni, Konstantinos Kaparis, and Adam N. Letchford. Cutting planes for rlt relaxations of mixed 0–1 polynomial programs. *Mathematical Programming*, 151:639–658, 01 2015.

[16] Kimmo Soramäki and Samantha Cook. Sinkrank: An algorithm for identifying systemically important banks in payment systems. *Economics: The Open-Access, Open-Assessment E-Journal*, 7:1, 2013.

[17] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual.* 2022.

[18] Thorsten Koch, Tobias Achterbergm, Erling Andersen, Oliver Bastert, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenica Salvagnin, Daniel E. Steffy, and Kati Wolter. Miplib 2010: Mixed integer programming library version 5. *Mathematical Programming Computation*, 3:103–163, 07 2011.