Settling Debts Efficiently: Zero-Sum Set Packing

Achknowledgements

I would like thank my thesis adviser, Professor Michael Mitzenmacher for his endless advice that made this thesis possible. Thank you for encouraging me to pursue this topic, inspiring me to think about new algorithms and complexity results, helping me spot mistakes in my proofs, and reading over my drafts. All of this was worth it just to hear you say: "That's pretty cool!" to my results.

I would also like to thank my thesis readers Professor Jelani Nelson and Professor Madhu Sudan for generously agreeing to be my thesis readers as well the helpful conversations we had that motivated the algorithms of this thesis. In particular, I would like to thank Jelani for inspiring me in CS 124 freshman year, which made me want to more coursework and this thesis in theoretical computer science.

Finally, this thesis would not have been possible without the support of my friends. Thank you to Tiffany Wu and Christina Chen for proofreading my draft, Collegium's thesis fairies for all the snacks, and to all my friends who have wished me good luck on this thesis.

The topic of this thesis is inspired by a real life algorithmic problem that I came up with in the summer of 2015. At the time, it was just a problem I would think about for fun or tell my friends to get their thoughts. I'm proud of having spent so much of the past 6 months fully digging into this problem and turned a dinner conversation topic into a senior thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

A common problem that arises within a group of friends or roommates is how to settle debt amongst themselves when often times restaurants only accept one credit card or people buy communal items for the group. The most natural way of solving this problem is to use peer-to-peer payment methods to settle debt after every transaction. If you and 4 other roommates go out to eat and the bill comes out to $100, then whoever pays the bill should receive $20 of Venmo, Paypal, Messenger Pay...etc. from the other 4 people.

But if your group of friends often need to pay each other, it can become extremely cumbersome for everyone to pull out their phones and pay each other after every expense. Some people may not trust these peer-to-peer payment systems and so would prefer to minimize the number of payment they make or receive. A simple way of solving this problem is to keep track of all the accumulated debt on a spreadsheet. Instead of settling the debt after each expense is incurred, the group can settle the debt all at once later on, perhaps at the end of the semester. This saves time because only one person needs to open the spreadsheet and record everyone's balance. Figure 1 is an example of what such a spreadsheet would look like.

If an expense is y dollars, and it is split along n people, then everyone should have to pay $\frac{y}{n}$ dollars. However, if only one person covered the bill, then that person overpaid by $\frac{n-1}{n} \cdot y$. The remaining $n-1$ people underpaid by $\frac{y}{n}$ because that's how much they should have paid, but in actuality they paid 0. The numbers in the spreadsheet represent a person's debt, so positive numbers mean that they owe money to the group, while negative numbers mean they are owed money from the group.

Expense	Total Cost	Evan	Cherie	Sam	Ignacio	Sean
Common Room Carpet	$60	−48	12	12	12	−12
Dinner at Gyukaku	$125	25	25	25	−100	25
Waffles at Zinneken's	$50	10	10	−40	10	10
Movie Tickets	$68	17	−51	17	17	0
Total Balance:	$303	4	−4	14	−61	47

Figure 1: Example expense sheet

For example, with the movie tickets expense, only Evan, Cherie, Sam and Ignacio went to the movies so Sean's column gets a 0 because he did not participate in this expense. Cherie paid $68 for 4

tickets. She should only have paid $\$68/4 = \$17$, so she overpaid by \$51. Thus, we write a -51 in her column because she is owed 51 dollars. Evan, Sam and Ignacio each underpaid by \$17 so they get a 17 in column for the movie tickets expense, representing that they have accrued a debt of 17 dollars.

So at the end of the semester, each of the 5 people has a positive or negative balance. Having a positive balance means that you owe money to the group, but not to any one person in particular. Likewise, having a negative balance means that you are owed money from the group, but again not from any one particular person. In Figure 1, the last row represents the final balance or debt of the 5 people, with red meaning they owe money and green meaning they are short money. We will use the words debt and balance interchangeably.

1.2 Definitions and Objective

The flexibility of having no names attached to peoples' debt motivates the question of who should pay whom to settle the debt efficiently. To reduce the inconvenience to the group as a whole, our objective is to minimize the number of transactions needed in order to settle all the debt. When one person sends some amount of money to another person via Venmo, Paypal...etc, that is considered one transaction. Each transaction requires the sender to pull out their phone, and the receiver to accept the transfer, which is quite cumbersome and thus we want to avoid this as much as possible. When we refer to the debt of a particular person, we are always referring to their final debt, at the end of the semester after all expenses have been accounted for. The goal is to minimize the overall burden to the group by minimizing the total number of transactions needed for everyone to have their debts resolved.

Definition 1.2.1. A **transaction** is a triple (s, r, a) representing the sender, receiver and amount of the transaction. For example, $(1, 2, 10)$ would imply that agent 1 sends 10 dollars to agent 2. For notational convenience, we will consider $a < 0$ to mean that money was sent from r to s and thus $(1, 2, 10)$ would be equivalent to $(2, 1, -10)$. Sending 10 dollars from person 1 to person 2 is equivalent to person 2 sending -10 dollars to person 1.

Definition 1.2.2. A set of transactions $\{(s_j, r_j, a_j)\}_{j=1}^{t}$ is said to **settle** the debts $\{x_1, x_2 \dots x_n\}$ if

$$\sum_{j:s_j=i} a_j - \sum_{j:r_j=i} a_j = x_i \qquad \forall i \in \{1, 2, 3 \dots n\} \tag{1.1}$$

When $s_j = i$, that means i is the sender of money, and thus i's debt decreases by a_i. If i's balance was positive, meaning they owed money to the group, now it is less negative after making this payment. When $r_j = i$, that means i is the receiver of money and thus i's balance increases (or becomes less negative). Therefore, if $x_i > 0$, then that means that through these transactions, agent i must have sent x_i more dollars than he received, which would make his balance 0. Likewise, if $x_i < 0$, then the agent must have received x_i more dollars then he sent, which would also make his balance 0.

Definition 1.2.3. The **Debt Settling Problem** is the problem where you are given as input $X = \{x_1, x_2, x_3 \dots x_n\}$, which represent the final debt of n people (e.g. the last row of Figure 1), with the properties that $x_i \in \mathbb{Z}$ and $\sum_{i=1}^{n} x_i = 0$. The objective is to find a set of t transactions which settles all the debt, with t as small as possible.

1.3 Solutions

We will first look at the most natural way of finding the minimum number of transactions. We then look at a way of settling all the debt in a smaller number of transactions and generalize what is special about this situation that allows us to find this more efficient solution.

1.3.1 Natural Solution

Going back to the motivating example with the roommates, no matter what the final balances of the 5 people are, we can always settle the debt in 4 transactions. Simply designate someone as the bank and everyone can pay that person or receive from that person. For example, in the above example, we can designate Evan as the bank and then the payments would be:

$$\begin{aligned} \text{Evan} &\xrightarrow{\$4} \text{Cherie} \\ \text{Sam} &\xrightarrow{\$14} \text{Evan} \\ \text{Evan} &\xrightarrow{\$61} \text{Ignacio} \\ \text{Sean} &\xrightarrow{\$47} \text{Evan} \end{aligned}$$

More generally, this means that for any group of n people with some zero-sum debt among themselves, we can always settle the debt in $n-1$ transactions. Here, we use $s_i \xrightarrow{a_i} r_i$ to represent the transaction (s_i, a_i, r_i) where a_i dollars is sent from s_i to r_i because it is more intuitive to read.

Lemma 1.3.1. *For the debt settling problem with n agents, it is always possible to find a set of $n-1$ transactions which resolves everyone's debt.*

Proof. Label agent 1 as the bank. The $n-1$ transactions are:

$$\{(2,1,x_2),(3,1,x_3)\dots(n,1,x_n)\}$$

Recall in a transaction if the amount is negative, then that is equivalent to money being sent from the receiver to the sender. For example, if $x_2 = -10$, then x_2 is owed 10 dollars, so we have the transaction $(1,2,10)$ meaning that agent 1 will pay agent 2 \$10. We need to check that equation 1.1 for the debt being settled is satisfied:

For $i \neq 1$, we have that $\sum_{j:r_j=i} a_j = 0$ because agent 1 is the only receiver in any of these transactions. In addition, $\sum_{j:s_j=i} a_j = x_i$ because the only term in the sum comes from the transaction $(i,1,x_i)$. Plugging into equation 1.1, we get:

$$\sum_{j:s_j=i} a_j - \sum_{j:r_j=i} a_j = x_i - 0 = x_i$$

For $i=1$, we know that $\sum_{j:r_j=i} a_j = \sum_{j=2}^{n} x_i$ and that $\sum_{j:r_j=i} a_j = 0$. This is because agent 1 is the receiver of $x_2, x_3, x_4 \dots x_n$ dollars from agents $2,3,4,\dots,n$ respectively. Agent 1 is never the sender. Plugging this into equation 1.1, we get:

$$\sum_{j:s_j=i} a_j - \sum_{j:r_j=i} a_j = 0 - \sum_{j=2}^{n} x_j = x_1$$

because the sum of the x_i's is always 0. Therefore, after these $n-1$ transactions, each of the agents has their debt settled. □

Alternatively, another natural solution that works just as well is for all everyone to pay everyone else in a line. That would mean we have something like Evan paying Cherie, Cherie paying Sam, Sam paying Ignacio, and Ignacio paying Sean. Both of these method of paying take $n-1$ transactions and will be referred to as the natural solution from here onwards.

1.3.2 More Efficient Solution

We just saw that it is always possible to find a set of $n-1$ transactions to settle all the debt. However, this is not necessarily the *minimum* number of transactions necessary to settle all the debt. Going back to the motivating example from Figure 1, we can settle the debt in just 3 transactions:

$$\begin{aligned} \text{Evan} &\xrightarrow{\$4} \text{Cherie} \\ \text{Sam} &\xrightarrow{\$14} \text{Ignacio} \\ \text{Sean} &\xrightarrow{\$47} \text{Ignacio} \end{aligned}$$

It is easy to verify that everyone's debt is settled. What about the final balances $X = \{-4, 4, -14, 61, -47\}$ allows us to settle the debt in fewer than $n-1$ transactions? It turns out that it is precisely because X can be broken up into 2 disjoint pieces $X_1 = \{-4, 4\}$ and $X_2 = \{-14, 61, -47\}$ such that each piece has sum zero. The more zero-sum pieces we can partition X into, the fewer transactions we need to settle all the debt.

The goal of this thesis is to algorithmically explore this question of how to settle debt efficiently. We show that this problem reduces to a special case of the *set packing* problem and prove that this special case is also NP-Complete. We explore approximation algorithms for finding the minimum number of transactions and explore interesting variations to this problem as well as their computational complexity.

Chapter 2

Reductions

2.1 Reduction to Zero-Sum Set Packing

In this section, we show how the debt settling problem reduces to a variation of the set packing problem. From the motivating example in the previous chapter, we already observed that the more zero-sum sets we could partition X into, the fewer transactions are needed. Here, we will formalize that observation.

2.1.1 Relation to Finding Zero-Sum Sets

Theorem 2.1.1. *Given debts $X = \{x_1, x_2, x_3 \dots x_n\}$ with $x_i \in \mathbb{Z}, \sum_{i=1}^{n} x_i = 0$, there exists a sequence of $n-k$ transactions to settle all the debt if and only there exists some way to partition X into k disjoint sets of sum 0.*

Proof. First, we show that if such a partitioning of X into k disjoint sets exists, then we can settle the debt in $n-k$ transactions. Let $Y_1, Y_2 \dots Y_k$ be a partitioning of X such that each $\sum_{x \in Y_j} x = 0$. By definition of a partitioning, all the Y_j's are disjoint from each other and $\bigcup_{i=1}^{k} Y_j = X$. For a given Y_i, we can settle the debt in $|Y_i| - 1$ transactions by having everyone pay each other in the natural way described by lemma 1.3.1. That solution involved choosing an arbitrary person to be the bank and have everyone pay or receive from them. This settles the debt in

$$\sum_{i=1}^{k} (|Y_k| - 1) = n - k$$

transactions, which is exactly what we wanted to show.

For the other direction, we show how can we turn a sequence of $n-k$ transactions into a partitioning of X into k sets such that the sum of debts in each set is 0. We will partition the people into k groups in the following manner:

Start with an undirected graph $G = (V, E)$ with n singleton nodes representing each of the n people. $V = \{v_1 \dots v_n\}$ while E starts out as the empty set . For each transaction (s_i, r_i, a_i), add the undirected edge (s_i, r_i) to E. Do this for all $n - k$ transactions. We started with n connected components of the graph (i.e. the n singleton nodes) and with each iteration, the number of connected components could possibly stay the same, or go down by 1 because two components got unioned into 1. Therefore, after $n - k$ such iterations, the number of connected components

in G is somewhere between $n-1$ and $n-(n-k)=k$ inclusive. Call that number m and let $S_1, S_2, \ldots, S_m \subset \{1, 2, \ldots n\}$ be the indices of the vertices in each of the m connected components.

Lemma 2.1.2. *For each S_i, we have $\sum_{s \in S_i} x_s = 0$.*

S_i carries the indices of the people who are in the ith connected component. The claim is that the sum of the debts of the people in any of the connected components is 0.

Proof. The proof relies on the fact that we only connected two components together if there is a transfer of money between the two groups. Therefore, we know that there are no transactions between any of the people in S_i and the people in $\{1, 2, 3 \ldots n\} - S_i$. This means that money has been conserved in this group of people. If everyone in S_i has debt 0 at the end of these $n-k$ transactions, then their initial debts must sum to 0 or else there wouldn't be a way to make everyone break even.

We will use the definition of what it means for a set of transactions to settle all the debt, equation 1.1. For each $s \in S_i$, we have that:

$$\sum_{j:s_j=s} a_j - \sum_{j:r_j=s} a_j = x_s$$

We can sum up all the above equations for $s \in S_i$. This gives us:

$$\sum_{s \in S_i} \sum_{j:s_j=s} a_j - \sum_{s \in S_i} \sum_{j:r_j=s} a_j = \sum_{s \in S_i} x_s \tag{2.1}$$

For every transaction (s_j, r_j, a_j) where $s_j, r_j \in S_i$, we know that a_j will appear once in equation 2.1 in the first double summation when $s = s_j$ and once in the second double summation when $s = r_j$. Since there are no transactions where one end is in S_i and the other end is not in S_i, we have a perfect matching between terms in the first double summation and the second, which means that

$$\sum_{s \in S_i} \sum_{j:s_j=s} a_j - \sum_{s \in S_i} \sum_{j:r_j=s} a_j = 0$$

Hence, $\sum_{s \in S_i} x_s = 0$.

□

With this lemma, we have shown that $S_1 \ldots S_m$ is actually a partitioning of $\{1, 2, \ldots n\}$ into m subsets such that the sum of the debts of the people in each S_i is 0. Thus, define $T_i = \{x_s : s \in S_i\}$ and then $T_1, \ldots, T_m$ is a partitioning of X into m sets each of whose sum is 0. Our goal was to find a partitioning of X into k sets of sum 0, but since we already argued that $k \leq m \leq n-1$, we have that $T_1, T_2 \ldots T_{k-1}, \bigcup_{j=k}^{m} T_j$ is a valid partitioning of X into k zero sum sets. Notice that the sum of the elements in $\bigcup_{j=k}^{m} T_k$ is 0 because it is the union of a bunch of zero sum sets. Thus, we have shown that if there exists a sequence of $n-k$ transactions which settles all the debt, then it must be possible to partition X into k sets each of sum 0. □

With this theorem, we have shown that finding the smallest set of transactions is equivalent to finding the maximum number of zero-sum sets that you can partition X into. Equivalently, we can formulate this problem as trying to find the maximum number of disjoint subsets of X such that each subset has sum 0, which is a special case of the maximal set packing problem (referred to as just the set packing problem).

Definition 2.1.1. In the **(Maximal) Set Packing** problem, you are given a universe of elements U as well as subsets $S_1, S_2 \dots S_n \subset U$. The goal is to choose as many of the S_i's as possible such that the chosen S_i are mutually disjoint.

Remark: This is a well known NP-complete problem [1].

Definition 2.1.2. In the **Zero-Sum Set Packing** problem, you are given as input $X = \{x_1, x_2 \dots x_n\}$ where $x_i \in \mathbb{Z}$ and $\sum_{i=1}^{n} x_i = 0$. The goal is to find $S_1, S_2 \dots S_t \subset X$ such that $\sum_{s \in S_i} s = 0$ for all i, $S_i \cap S_j = \emptyset$ for all $i \neq j$ and t is as large as possible.

Finding the maximum number of zero sum sets that you can partition X into is equivalent to finding the optimal zero-sum set packing. We saw from theorem 2.1.1 that the debt setting problem and the zero sum set packing problem reduce to one another. We can use the debt settling problem to solve the zero-sum set packing problem and vice versa. The zero-sum set packing problem is a special case of the NP-complete set packing problem. Our goal in the next section is to show that the zero-sum set packing problem is also NP-complete and thus the debt settling problem is NP-complete.

2.2 NP-Completeness

2.2.1 Proof

In this section, we will show that the debt settling problem is NP-Complete. Since we have shown that the debt settling problem reduces to the zero-sum set packing problem, we will also have shown that that is NP-complete. Our reduction will rely on the fact that in order to solve the debt settling problem, we have to find subsets of sum 0, but doing so is related to the NP-complete subset sum problem [1].

Definition 2.2.1. In the **Subset Sum** problem, you are given a set of numbers $X = \{x_1, x_2 \dots x_n\}$ and are asked whether there exists $S \subset X$ such that $|S| > 0$ and $\sum_{s \in S} s = 0$.

Note: In the subset sum problem, we certainly do not require that the sum of all the elements of X be 0. In the debt settling problem, we do require X to be zero-sum.

Theorem 2.2.1. *The debt settling problem is NP-Complete.*

Proof. Given any instance of the subset sum problem, we will show that it can be reduced to an instance of the debt settling problem. Suppose we are given $X = \{x_1, x_2 \dots x_n\}$ where the sum of the x_i's is not necessarily 0 and let $s = \sum_{i=1}^{n} x_i$. We can convert it into an instance of the debt settling problem with $n + 1$ people whose debts are $\{x_1, x_2 \dots x_n, -s\}$. This fulfills the condition that the sum of all the people's debts is 0.

The claim is that there exists a subset of sum 0 in X if and only if the debt on $X \cup \{s\}$ can be settled in at most $n - 2$ transactions. By Theorem 2.1.1, we know that this is equivalent to asking whether these exists a partioning of $X \cup \{s\}$ into 2 disjoint subsets of sum 0. If X has a subset of sum 0, call it X', then X' and $(X \cup \{s\}) - X'$ would be a valid partioning.Either X' or $(X \cup \{s\}) - X'$ must not contain s' and hence that set is a subset of X and also has sum 0. If it is not possible to settle the debt in $\leq n - 2$ transactions, then it must be that the only zero-sum subset of $X \cup \{s\}$ is the set itself, and hence X has no zero-sum sets. This completes the proof that the subset sum problem reduces to the debt settling problem, and thus also to the zero-sum set packing problem. □

Corollary 2.2.1.1. *The zero-sum set packing problem is NP-Complete*

2.2.2 NP-Complete Pipeline

The debt settling problem and the zero sum set packing problem are interesting because they are not only NP-Complete, but they can be viewed as the composition of two NP-Complete problems. We are given X, the set of debt, but we are not given which subsets of X are of sum 0. We need some efficient way of finding all the zero-sum subsets of X. After we get those zero-sum sets, then we feed that into the set packing problem and that gives us a solution to the zero-sum set packing problem.

Figure 2: The NP-Complete Pipeline

This kind of two-layered NP-complete problem motivates the following questions related to algorithms and complexity:

Algorithms: One of the bottlenecks for finding an efficient algorithm for the zero-sum set packing problem is that we are not given the zero-sum sets as part of our input. Finding all these zero-sum sets, at least through the brute-force way, will take $O(2^n)$ time. If we want to find a polynomial-time approximation algorithm for the zero-sum set packing problem, we surely cannot find *all* the zero-sum subsets. Do we really need to find *all* zero sum subsets of X in order to find a *good* zero-sum packing of X? Will feeding a subset of the zero-sum sets into the set packing algorithm guarantee a good solution most of the time?

Complexity: Suppose we removed the bottleneck of finding all the zero-sum subsets. This means that as part of your input, you are given all the zero-sum subsets. Then, the problem simply becomes a set packing problem on those given zero-sum subsets. If these were arbitrary subsets, the problem is just the set packing problem and is thus NP-Complete. However, zero-sum subsets have interesting properties that are not true for subsets in general. These will be discussed in the next section. Is the set packing problem still NP-complete even when the sets we are dealing with have these special properties?

2.3 Zero Sum Subsets

In this section, we will explore some of the properties of zero-sum subsets. These properties will be useful when we design algorithms for approximating the zero sum set packing problem and the debt settling problem in chapter 3. These properties suggest perhaps the set packing problem is easier when the sets we can choose from are all the zero-sum sets of X. In chapter 4, we will explore whether zero-sum set packing is fundamentally less difficult than regular set packing.

2.3.1 Structure of Zero-Sum Subsets

Let $X = \{x_1, x_2 \dots x_n\}$ be the debts (so that the sum of all the x_i is 0). Let $S = \{S_1, S_2, \dots S_m\}$ be a collection of all the zero-sum subsets of X. Then, the following properties are true:

- (Empty Set) $\emptyset \in S$

- (Full Set) $X \in S$
- (Disjoint Union) For any two sets $S_i, S_j \in S$, if $S_i \cap S_j = \emptyset$, then $S_i \sqcup S_j \in S$.
- (Subset Difference) For any two sets $S_i, S_j \in S$ with $S_i \subset S_j$, we have that $S_i - S_j \in S$.

These are the most intuitive properties of zero-sum sets, that subsets are closed under disjoint union. However, are these properties sufficient for us to say anything about whether zero-sum set packing is fundamentally different from the general set packing? Are they a full characterization of zero-sum sets?

2.3.2 Additional Structure

In this section, we will show that zero-sum subsets have an additional property that is not covered by the disjoint union and subset difference properties from above. Using a few zero-sum subsets, we will deduce the existence of another zero-sum subset. The catch is that this new zero-sum subset cannot be made by using any of the above 4 properties on the given zero-sum subsets!

$$X = [-3, -4, -2, 9, -7, 8, 1]$$

Note that these number don't add up to 0, but that is okay because we're just trying to demonstrate the properties of zero-sum subsets outside of a debt settling context. Let three of the zero-sum subsets of X be $S_1 = \{-2, -3, -4, 9\}, S_2 = \{-2, -7, 9\}, S_3 = \{-2, -7, 1, 8\}$. The below figure shows the zero-sum sets of the bolded elements. The elements within a circle have sum equal to 0.

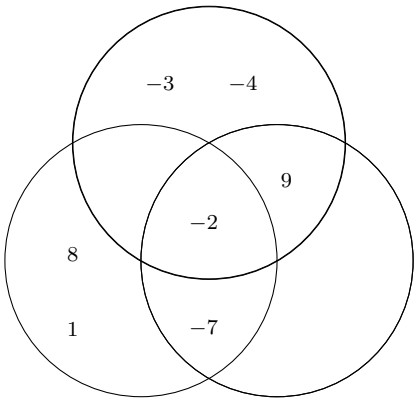


Figure 3: Organization of the Zero-sum Sets

The goal here is to show that S_1, S_2, S_3 being zero-sum imply that there exists another zero-sum set S_4. S_4 cannot be made by applying any of the 4 properties above to S_1, S_2, S_3. We can write the following equations.

$$\begin{aligned}(-3) + (-4) + (-2) + 9 &= 0\\ (-2) + (-7) + 9 &= 0\\ (-2) + (-7) + 8 + 1 &= 0\end{aligned}$$

If I take $(1) - (2) + (3)$, without doing any arithmetic besides $a + (-a) = 0$ and $a - a = 0$, we get:

$$(-3) + (-4) + (-2) + 8 + 1 = 0$$

Thus, we also have $S_4 = \{-2, -3, -4, 1, 8\}$. This means that the existence of the zero-sum subsets S_1, S_2, S_3 somehow implied the existence of S_4 even though S_4 cannot come about through the 4 properties above.

The purpose of this example is to show that there is more to zero-sum subsets than just the 4 properties listed in the previous section. There are ways that we can start with a few primitive subsets and find more primitive subsets. This really begs the question: To what extent do features of zero-sum subsets help us in the set packing problem?

Chapter 3

Algorithms

In the previous chapter, we introduced the debt settling problem and the zero-sum set packing problem and showed that they reduce to each other. In this chapter, we will explore approximation algorithms for the two problems. Even though these two problems reduce to one another, approximations are not preserved in the reduction because $n - k$ transactions correspond to a set packing of size k.

We will first see an exponential time algorithm which finds solves the zero-sum set packing and debt settling problems exactly. Then, we explore approximation algorithms for the zero-sum set packing problem. We propose an algorithm that achieves a approximation bound with high probability over all possible inputs generated according to a probabilistic model. Finally, we will explore deterministic polynomial-time approximation algorithms for debt settling problem to find the minimum sequence of transactions.

3.1 Exponential Exact Algorithm

Before we present this exponential time algorithm for the zero-sum set packing problem, we will first prove a few interesting properties of subsets of size 1 and 2. This properties allow allow us to do some pre-precessing to our list of debts before we run our exponential exact algorithm. Although it does not give us any theoretical improvements, these are properties that only hold for zero-sum subsets and thus are important to consider when analyzing the computational complexity of this problem, especially in relation to the general set packing problem.

3.1.1 Subsets of Size 1 and 2

In this section, we will show that we can greedily remove all zero-sum subsets of size 1 and 2 from X. Let $X = \{x_1, x_2 \dots x_n\}$ be the debts as always, with the sum of the elements of X being 0. Our goal is to find the maximum number of disjoint zero-sum subsets of X.

If there exists a zero-sum subset of size 1, then we should greedily add it to the set packing. A zero-sum subset of size 1 can only be the set containing only 0. Suppose $Y = \{0\}$ and is not used in the optimal solution. Then Y is subset of some zero sum set Z that was used. We can split Z into Y and $Z - Y$ and we would get a valid zero sum set packing of larger size! This is a contradiction to the assumption that Y was used in not used in the optimal solution. Therefore, we can greedily take all the singleton 0's out of the input.

Lemma 3.1.1. *If there exists a zero-sum subset of size 2, then we should greedily add it to the optimal set packing.*

Proof. Without loss of generality, suppose that $x_1 + x_2 = 0$ and define $X' = \{x_3, x_3 \dots x_n\}$. Let $S_1, S_2 \dots S_m \subset X$ be the optimal packing of zero-sum subsets of X. Since X itself is zero-sum, it must be the case that $\bigcup_{i=1}^{m} S_i = X$. We know that x_1 and x_2 must be in two different S_i's because if they were in the same S_i, then we could partition S_i into the zero-sum sets $S_i - \{x_1\} - \{x_2\}$ and $\{x_1, x_2\}$, which contradicts the fact that our collection of S_i's was optimal. Therefore, without loss of generality, assume that $x_1 \in S_1$ and $x_2 \in S_2$. This implies that another possible packing that achieves the same number of disjoint subsets m is to have:

$$\{\{x_1, x_2\}, (S_1 - \{x_1\}) \cup (S_2 - \{x_2\}), S_3, S_4 \dots S_m\}$$

This means that we can greedily add $\{x_1, x_2\}$ into our packing, recurse on X' and we will still be on-track with an optimal solution that also takes this zero sum set of size 2. □

3.1.2 Description

First, we can filter out all 0's and then use lemma 3.1.1 to greedily take as many size 2 zero-sum set as possible. Then, we will run a dynamic programming algorithm whose state is the subset of the n numbers that have not been packed yet. At each step, if there are m unpacked elements, we know that these m elements all together form a zero-sum subset, and thus we loop through all subsets of these m elements of size $3, 4, \dots \lfloor m/2 \rfloor$ and check to see if any of them are of sum 0. If so, we try to take that subset of elements and recurse on what's left. We take the branch of the recursion with the largest size packing. If no such subset is of sum 0, then we are at the base case where we return that the optimal packing of these m elements is just the entire set.

Algorithm 1 Exponential time algorithm for finding best zero-sum set packing

```
1:  procedure BestPartition(subset, debts)
2:      subset ← subset of vertices I'm looking at
3:      debts ← array of debt values
4:      return ← the best partitioning of subset
5:  Base Case:
6:      if susbet is empty then return [ ]
7:  Recursive Case:
8:      solutions ← [ ]
9:      for s ⊆ subset and 3 ≤ |s| ≤ len(subset)/2 do
10:         if subset is valid (has sum 0) then
11:             solutions.append ← [s] + BestPartition(subset − s, debts)
        return element in solutions with the largest length.
```

3.1.3 Analysis

Theorem 3.1.2. *Algorithm 1 runs in $O(3^n)$ time.*

Proof. We analysis the complexity of this algorithm by the number of states times the time it takes to process each state. There are 2^n possible subsets of *debts* that the algorithm can take as input. For each input, we iterate through all the subsets of the input subset, finding the one which gives us the maximum number of partitions. This can take 2^n time as well. Therefore, a crude upper

bound for the run-time seems to be $O(4^n)$.

However, out of the 2^n possible inputs to the function, there are $\binom{n}{i}$ of them of size i. On line 9 of the pseudocode, we loop through all the subsets of size less than $i/2$, and so that takes $O(2^i)$ time. Therefore, a more precise run-time of this algorithm is:

$$O\left(\sum_{i=1}^{n} \binom{n}{i} 2^i\right) = O(3^n)$$

Therefore, we have a $O(3^n)$ exact algorithm that solves the zero-sum set packing problem using dynamic programming. Note that removing lemma 3.1.1 does not give us any theoretical increases in run-time, but in practice, they are a great way to trim the list of debts before being fed into this exponential time algorithm. □

3.2 Approximation Algorithm for Zero-Sum Set Packing

3.2.1 Description

Our approximation for zero-sum set packing is very intuitive. Choose a constant $k \geq 3$. Given X like in our set-up as always, we can find all the zero-sum subsets of size k in $O(n^k)$ time, which is polynomial for fixed k. Then, given those $O(n^k)$ subsets, we can apply a polynomial time approximation algorithm for the k-set packing problem. The k-set packing problem is the set packing problem where every set you can choose is of size $\leq k$.

It is important to know that it is no longer true that we can greedily add all zero sum subsets of size 2 to our solution. Lemma 3.1.1 is no longer true. Here is a simple counter-example:

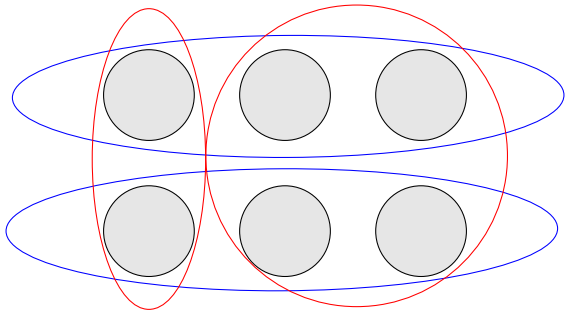


Figure 4: Counterexample for Lemma 3.1.1

Suppose we had $k = 3$ in this situation. Each circle represents an element and the zero-sum sets are marked by the ellipses. Then, if we greedily took the set of size 2 on the left, then we would not be able to take any more sets. The optimal solution is would be to take the 2 blue horizontal sets. Therefore, for our approximation algorithm, we cannot greedily throw away the sets of size 2 anymore. If we were allowed to use zero sum sets of any size, then we would be able to find the red set of size 4 in the example above, but if we are restricted to sets of up to a certain size, then

we cannot be so greedy. We can, however, still remove all zero-sum sets of size 0 because the same argument still applies.

3.2.2 Intuition

There is pretty good intuition as to why this algorithm should do well in practice. If we are trying to find the most number of non-overlapping zero-sum subsets as possible, smaller sets are going to be more helpful than larger sets. A large set of sum 0 is not as likely to be used in the final *optimal* set packing as a smaller set. Therefore, it seems reasonable that using only the smaller sets, we can find a pretty good set packing.

We have no guarantee as to whether there will even exist zero-sum subsets of size $\leq k$. It is entirely possible that all the zero-sum subsets are of size $> k$. Therefore, we cannot get a deterministic approximation algorithm because an adversary could simply feed us a packing which uses $\frac{n}{k+1}$ disjoint subsets of size $k+1$, and we would not find any subsets, so the approximation ratio can be as arbitrarily bad. The goal is to show that if we inputs from the following probabilistic model, then this kind of worst case situation will not happen very often. This is very difficult to show mathematically, so we will offer some experimental evidence.

3.2.3 Probabilistic Model

Before we discuss the approximation algorithms used, let's first talk about how we generate test cases to run our exact and approximation algorithms on. Let n be the number of people, and let m be the mean amount for each expense incurred by the group.

The amount of money for each expense will be normally distributed with mean m and standard deviation $\frac{m}{2}$. The number of expenses incurred will be fixed at $5n$. For each expense, we alternate the agents paying in a fair way, so transaction i out of $5n$ is paid by agent $i \mod n+1$. This guarantees that each of the n agents labeled $1, 2 \ldots n$ will pay exactly 5 times. Let $Y \sim N\left(m, \frac{m^2}{4}\right)$ be a particular expense. Then, the person who pays will place a $-\frac{n-1}{n} \cdot Y$ in their column of the expense sheet, while everyone else will put a $\frac{1}{n} \cdot Y$ in theirs.

Just given this setup, there seem to be a couple problems, namely that Y not being an integer and Y being negative. Here's how we deal with those problems:

- **Integers:** The amount of debt or surplus incurred by each person for a given transaction is kept as a floating point number. At the end of the time period, when each person's debt is aggregated, we round everyone's final debt to the nearest integer. This makes it so that the total sum of all the debts is integral and is certainly between $-\frac{n}{2}$ and $\frac{n}{2}$. If this sum is non-zero, then we need to force the sum to be 0 in order for it to even be possible to resolve all the debt. Suppose this sum is c. If $c > 0$, then we choose c of the n people uniformly at random to lose a dollar of debt (-1 to their balance). This makes the sum go down by c and thus the sum of everyone's debts is 0. Conversely, if $c < 0$, then we choose c of the n people uniformly at random gain a dollar of debt (+1 to their number), which will also make all the debts sum up to 0.
- **Negative:** There is roughly a 2.5% chance that Y will be negative for any particular expense. One might think that we would need to redraw Y because the amount incurred by an expense

must be positive. However, one can imagine a situation where the group receives a rebate or refund and one persons receives it, so he/she must split it with the entire group. Therefore, it is not realistic that a small percent of the time, we get some kind payout that needs to be split between the n people.

It is interesting to see that the final debts of the n agents follow a roughly multivariate normal distribution, except that we have made it so that all the numbers are integers.

3.2.4 Goals

Our goal will be to experimentally determine how often the following two situations arise when our inputs are generated according to the probabilistic model above. These two situations are interesting because they if they occur, then we have a provable bound on the best set packing with just sets of size k relative to the best set packing with unrestricted size sets. Let OPT represent the size of the maximum zero-sum set packing with sets of unrestricted size and let OPT_k be the size of the maximum zero-sum set packing with sets of size $\leq k$.

- k **Complete Packing:** It is possible to pack every element of X into disjoint zero-sum sets of size $\leq k$.
- k **Optimality:** Using only zero-sum sets of size k or less, the maximum number of sets that can be packed is equal to the maximum number of sets of unrestricted size that can be packed (i.e. $\text{OPT}_k = \text{OPT}$).

As we generate data from the probabilistic model from above, we will experimentally evaluate how often we have a k complete packing and how often we have k optimality.

Lemma 3.2.1. *Optimality implies complete packing.*

Proof. Suppose $\text{OPT}_k = \text{OPT} = c$. Let $S_1, S_2, \ldots, S_c \subset X$ be the optimal packing of zero-sum sets of size at most k. The claim is that $\bigsqcup_{i=1}^{c} S_i = X$, (i.e. the S_i's are a complete packing). This is true because if $S_1, S_2, \ldots, S_c$ is not a complete packing, then $\text{OPT} > c$ by taking the sets $S_1, \ldots, S_c$ and then finding the optimal zero-sum set packing of $X - \bigsqcup_{i=1}^{c} S_i$, which is also a zero-sum set. □

Corollary 3.2.1.1. *Given a random $X = \{x_1, x_2 \ldots x_n\}$ drawn according to the probabilistic model above and a fixed k, it is more likely that there exists a complete packing of X than that the optimal packing of X with $\leq k$ sized sets is the true optimal packing of X.*

This corollary falls very easily from the above lemma. If an event A implies B, then $P(A) \leq P(B)$.

Lemma 3.2.2. *Complete packing implies 2-optimality.*

Suppose we could find sets $Y_1, Y_2 \ldots Y_s$ such that $|Y_i| \leq k, Y_i \cap Y_j = \emptyset$ and each Y_i is a zero-sum set. If the Y_i's represent a complete packing (i.e. $\bigcup_{i=1}^{s} Y_i = X$), then we will show that $\frac{\text{OPT}}{\text{OPT}_k} \leq 2$. This does not imply, however, that s is within a factor of 2 from optimal because this particular packing of the Y_i's doesn't have to be the optimal way to pack zero-sum sets of size $\leq k$. It just means that whatever the optimal packing is with sets of size $\leq k$ is within a factor of 2 from the optimal packing with unrestricted sized sets.

Proof. The intuition behind this lemma is that if our optimal solution using only subsets of size $\leq k$ packs all of X, then any solution using sets of size $k+1$ can't do too well because it cannot

use a large number of sets of size $\leq k$.

Since it is possible to pack all the elements of X, we have that $\text{OPT}_k \geq \frac{n}{k}$. Any solution which packs every element of X must use at least $\frac{n}{k}$ sets and thus OPT_k must also be at least $\frac{n}{k}$. Now we want a lower bound on how good OPT can be. Suppose OPT uses a sets of size $\leq k$ and $\text{OPT} - a$ sets of size $> k$. We know that $a \leq \text{OPT}_k$ because otherwise OPT_k was not the optimal number of sets of size $\leq k$ that could be packed into X. We also have that $\text{OPT} - a \leq \frac{n}{k+1}$ because that's the maximum number of sets of size more than k that you could pack into X. Therefore, we have:

$$\begin{aligned}\text{OPT} &= a + \text{OPT} - a \\ &\leq \text{OPT}_k + \frac{n}{k+1}\end{aligned}$$

which implies that the approximation ratio is bounded by

$$\begin{aligned}\frac{\text{OPT}}{\text{OPT}_k} &\leq 1 + \frac{n}{\text{OPT}_k(k+1)} \\ &\leq 1 + \frac{n}{\frac{n}{k}(k+1)} \\ &\leq 1 + \frac{k}{k+1} \\ &\leq 2\end{aligned}$$

Thus, we have proven that if a complete packing of X with zero-sum sets of size k exists, then our OPT_k must be within a factor of 2 from OPT. □

For a given input $X = \{x_1, x_2 \ldots x_n\}$ drawn according to the probabilistic model discussed above. We are interested in the two situations from above for a fixed k.

- k **complete packing**: Does there exist a complete packing of X using disjoint zero-sum sets of size $\leq k$?
- k **optimality**: Is it true that the optimal packing with zero-sum sets of size $\leq k$ is equal to the optimal packing with zero-sum sets of unrestricted size?

Let p_1 the probability of having a k complete packing and p_2 be the probability of having k optimality happening. We know that $p_1 > p_2$ by lemma 3.2.1, where we showed that having an optimal solution implies the existence of a complete packing.

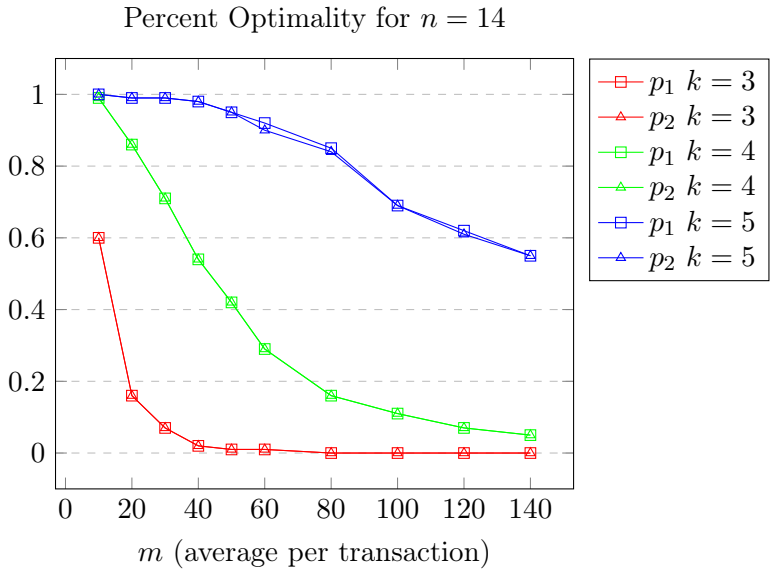
With probability p_2, we have that $\text{OPT} = \text{OPT}_k$ and with probability p_1, we have that $\text{OPT} \leq 2 \cdot \text{OPT}_k$. This makes sense because we are saying that with some smaller probability, the two optima are equal; with some higher probability, the two optima are off by at most a factor of 2. Therefore, if we can say that p_1 and p_2 are large, we will be able to say that only considering zero-sum sets of size $\leq k$ will often lead us to a solution that is either optimal or within a factor of 2 from the optimal packing with unrestricted sized sets.

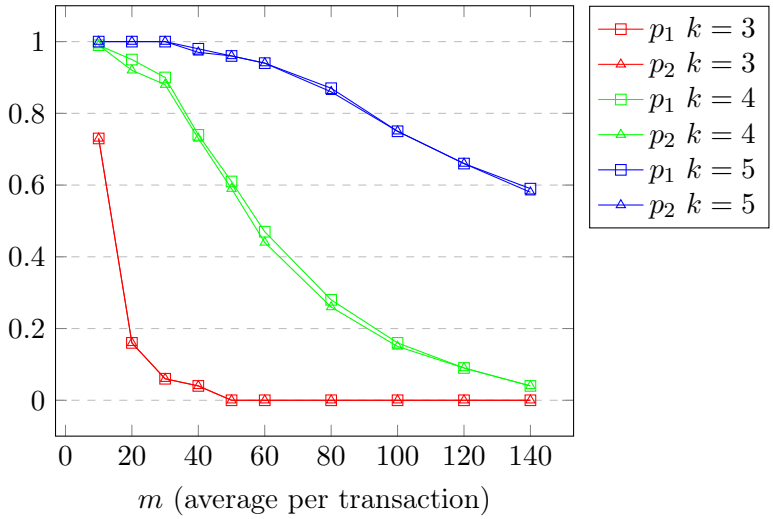
3.2.5 Implementation Results

Our goal here is to show that p_1 and p_2 are large most of the time. We will see in the data that it is usually sufficient to choose $k = 5$ and we will acheive 5-complete packing and 5-optimality with good probability. Recall n and m from the probabilistic model. There are n people, and $5n$

expenses, each normally distributed with mean m and standard deviation $m/2$.

In each of the following graphs, we fix a particular value of n and observe how the p_1 and p_2 changes as m and k vary. The y axis gives p_2, the percentage of time out of the 250 trials for each data point that the optimal solution when only using zero-sum sets of size $\leq k$ matched the optimal solution when using any sized zero-sum sets.





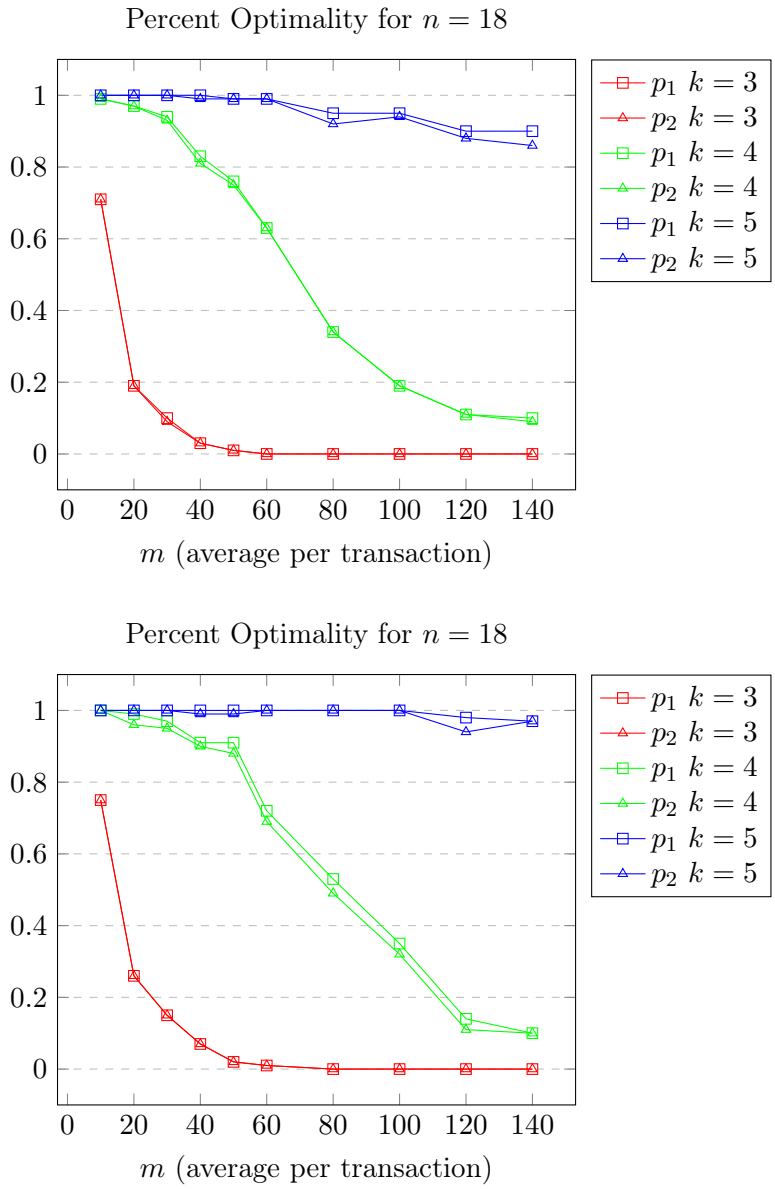


Figure 5: Data Showing p_1 and p_2 for various values of n, m, k

3.2.6 Analysis

Here are some of the interesting and perhaps obvious observations from the data and their intuitive explanations:

- **For a fixed value of n and k, larger m do worse.**
  As m gets larger, the magnitude of the values of X will increase. If each expense incurred the group is on average more expensive, we expect more variation in the final debts of all the people. For example, if $m = 10$, then we wouldn't expect any of the values of X to exceed 20 in magnitude, but if $m = 50$, many of the values of X will exceed 20 in magnitude.

  As the numbers of X get more and more spread out, it becomes harder to find smaller zero-

sum sets and therefore if n and k do not increase, then both p_1 and p_2 will decrease. We see this in the graph very clearly as all the lines slope downwards as m increases.

- **For a fixed value of m and n, larger k do better.**
  This is pretty intuitive because when k is large, we are capturing more possible subsets to use in our packing. In the extreme case where $k = n$, we will be allowed to use all of the zero-sum subsets of X. As k increases, we have more and more sets available at our disposal and thus are more likely to be able to both be able to pack all the elements and match the optimal solution.

- **For a fixed value of k and m, larger n do better.**
  When n is large, there are more zero-sum sets that we can possibly choose of a size $\leq k$. For example, if $n = 10, m = 50, k = 3$, then given how spread apart the numbers of X are, it is unlikely that we will sets of size 3 or less will be able to pack everything in X. However, if $n = |X| = 20$, suddenly the number of subsets of size 3 or less has grown quite a lot. With 10 numbers, it might be hard to find a way to break them into zero-sum groups of size ≤ 3, but with 20 numbers, there are more options of how you pair things together. Therefore, if m and k remain fixed, the more numbers there are, the more likely we sets of size k will be sufficient for finding the optimal solution.

3.2.7 Conclusion

We have seen from the data above, that once n is above 20, we can choose $k = 5$ and that gives us a 95%+ chance of successfully finding the optimal packing with only zero-sum sets of size ≤ 5. When $n \leq 20$, we can simply use the exponential exact algorithm running in $O(3^n)$ time and that would of course give us the exact solution. In practice, it is possible for this to run in a reasonable amount of time, especially given our heuristics of removing sets of size 1, 2 and going checking sets of size up to half the remaining unpacked elements. Once $n > 20$, the data above shows that using a $k = 5$ approximation to the zero-sum set packing problem does very well in practice. Therefore, for $n > 20$, almost certainly we will have $\text{OPT}_5 = \text{OPT}$. The best known 5-set approximation in the literature has approximation ratio $\frac{7}{3}$ [2]. Therefore, if we run the best 5-set packing approximation on all $O(n^5)$ zero-sum sets of size ≤ 5, this will give us a polynomial time algorithm for the zero-sum set packing problem that achieves a $\frac{7}{3}$ approximation factor with high probability (i.e. when $\text{OPT}_5 = \text{OPT}$). With even higher probability, this algorithm achieves a $\frac{14}{3}$ approximation (when $\text{OPT} \leq 2\text{OPT}$).

3.3 Approximations Algorithm for Minimum Number of Transactions

Recall Theorem 2.1.1 which said that if there exists a zero-sum set packing of with k sets, then there is a way to solve all the debt in $n - k$ transactions. In the previous section, we looked at an approximation algorithm for the zero sum set packing problem, which the debt settling problem directly reduces to. For that algorithm, although an adversary could supply an input that made our approximation arbitrarily bad, implementational evidence shows that on average, approximating the solution with only zero-sum sets of size at most k works well. In this section, we will look at approximation algorithms for the minimum number of transactions necessary to settle all the debt.

In this section, we will use approximation algorithms for the general k-set packing problem and show how they can be used to approximate the debt settling problem. The goal will be to show that if we have an α_k approximation for the k-set packing problem, then that translates to a certain approximation algorithm for the minimum number of transactions in the debt settling problem.

3.3.1 General Form

In this subsection, we will present the general recipe for the algorithms described later that achieve this deterministic constant approximation for the debt settling problem.

Our algorithm will be based on the idea of finding as many disjoint zero-sum sets as possible of size up to a fixed constant k. Recall from theorem 2.1.1 that there exists a sequence of $n - m$ transactions if and only if there exists a partitioning of X into m sets of sum 0. Our algorithm will try to find as many disjoint zero-sum subsets of X as possible that are of size $\leq k$, and we will show that this gives us a approximation bounds on the minimum number of transactions needed. Without loss of generality, we assume that 0 does not appear in $X = \{x_1, x_2, \dots x_n\}$ because all the people with 0 debt can be removed. **We assume for the rest of this chapter that X contains no zero-sum sets of size 1.**

We will use existing polynomial-time approximation algorithms for the k-set packing problem to try and find an approximately large packing of $\leq k$ sized zero-sum sets. The k-set packing approximation algorithm we will use was proposed by [2]. It uses a local search algorithm to prove a $\frac{k+2}{3}$ approximation to the maximum number of size $\leq k$ sets that we can pack.

Algorithm 2 Settles debt by using the k-set packing approximation

Our algorithm takes as input X, the list of debts for n people, and an integer k. We exhaustively find all $O(n^k)$ subsets of X of size $\leq k$ and filter only for the ones whose sum is 0. Then, we feed these $O(n^k)$ subsets into an approximation algorithm for the k-set packing problem. Suppose that approximation algorithm returns to us the sets $X_1, X_2 \dots X_m$ where each $X_i \subset X$ is zero-sum and all X_i are disjoint from one another. For each X_i, we will have the corresponding people to these debts pay each other in the naive way, settling the debt in $|X_i| - 1$ transactions. For any people remaining whose debt did not get packed into one of the X_i's, we will settle all those people's debts in the natural way.

Here is an example of how algorithm 2 works when $k = 3$. The nodes represent people while the arrows represent transactions. We use the notation from the upcoming theorem, where s_i is the

number of X_j of size i found from algorithm 2.

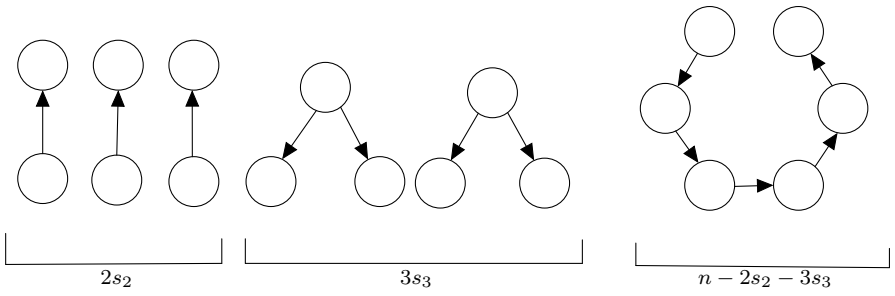


Figure 6: An Example when $k = 3$

Theorem 3.3.1. *If we use an exact algorithm for the k-set packing problem, then Algorithm 2 would be a* $\frac{k+1}{k}$ *approximation to the minimum number of transactions needed.*

Proof. Suppose we ran algorithm 2 as above. Let $s_1, s_2, \ldots s_k$ count the number of X_i's that are of size $1, 2, \ldots k$ respectively. Let $t_1, t_2, \ldots, t_k$ count the number of zero-sum sets of size $1, 2, \ldots, k$ that are packed in the optimal solution where we can use zero-sum subsets of unrestricted size. Given these definitions of the s's and t's, how many transactions does algorithm 2 use? What is the minimum number of transactions that the optimal solution with unrestricted sized sets uses?

For each of the s_j subsets of size j used, it takes $j - 1$ transactions to settle the debt for a total of

$$\sum_{j=2}^{k}(j-1)\cdot s_j \tag{3.1}$$

transactions. If we do this for each j, then the number of people left whose debt has not yet been settled is

$$\ell = n - \sum_{j=2}^{k} j \cdot s_j \tag{3.2}$$

Those people's debts can be settled in the natural way, which takes $\ell - 1$ transactions. Note that if $\ell = 0$, then we of course can settle the debt in 0 transactions, not -1 transactions. To remedy this, we will use an upper bound and say that if there are ℓ people remaining, then that debt can be settled in at most ℓ transactions. Therefore, adding up what we have from equations 3.1 and 3.2, we get:

$$\sum_{j=2}^{k}(j-1)\cdot s_j + n - \sum_{j=2}^{k} j\cdot = \; n - \sum_{j=2}^{k} s_j \tag{3.3}$$

Now, the question is what is a lower bound on the number of transactions of the optimal solution, which can use any sized zero-sum sets? For each of the t_j zero-sum sets of size j, the optimal solution takes $j - 1$ transactions.

$$\sum_{j=1}^{k}(j-1)\cdot t_j \tag{3.4}$$

For the remaining elements that are not packed into these sets of size $\leq k$, the best possible scenario for the optimal solution is if they can all be perfectly partitioned into zero-sum sets of size $k+1$. Each of these zero-sum sets of size $k+1$ would lead to k transactions. Therefore, after settling the debt for everyone whose debt was packed into a set of size $\leq k$, the number of additional transactions necessary must be more than

$$\frac{k}{k+1}\left(n - \sum_{j=2}^{k} j \cdot t_j\right) \tag{3.5}$$

Therefore, the optimal solution uses at least the sum of equations 3.4 and 3.5 number of transactions:

$$\sum_{j=2}^{k}(j-1)\cdot t_j + \frac{k}{k+1}\left(n - \sum_{j=2}^{k} j \cdot t_j\right) = \frac{k}{k+1}n - \sum_{j=2}^{k}\left(1 - \frac{j}{k+1}\right)t_j \tag{3.6}$$

The approximation ratio is calculated as equation 3.3 divided by 3.6. But we need to somehow relate the s's to the t's.

Lemma 3.3.2. *If the s's came about from an exact algorithm for the k-set packing problem, then*

$$\sum_{j=2}^{k} s_j \geq \sum_{j=2}^{k} t_j \tag{3.7}$$

Proof. Suppose for contradiction that $\sum_{j=2}^{k} s_j < \sum_{j=2}^{k} t_j$, yet our s_i's came about as a result of an exact $k-$set packing algorithm. Our exact algorithm says that $\sum_{j=2}^{k} s_j$ is the maximum number of sized $\leq k$ zero-sum subsets we can pack at once, yet there exists some other solution which uses *more* $\leq k$ sized sets, namely $\sum_{j=2}^{k} t_j$. This is a contradiction to the assumption that s_j's came about via an exact algorithm that gives us the optimal k-set packing. Therefore, we must have $\sum_{j=2}^{k} s_j \geq \sum_{j=2}^{k} t_j$. □

By a similar reasoning, we have the following corollary to lemma 3.3.2

Corollary 3.3.2.1. *If the s's came about from an α approximation for the k-set packing problem, then*

$$\sum_{j=2}^{k} s_j \geq \frac{1}{\alpha}\left(\sum_{j=2}^{k} t_j\right) \tag{3.8}$$

Using lemma 3.3.2 that we just proved, the approximation ratio, which is calculated by equation 3.3 divided by equation 3.6, is bounded by:

$$\frac{n - \sum_{j=2}^{k} s_j}{\frac{k}{k+1}n - \sum_{j=2}^{k}\left(1 - \frac{j}{k+1}\right)t_j} \leq \frac{n - \sum_{j=2}^{k} t_j}{\frac{k}{k+1}n - \sum_{j=2}^{k}\left(1 - \frac{j}{k+1}\right)t_j} \tag{3.9}$$

Our goal is to maximize the expression above, subject to the constraint that $t_j > 0$ for all j as well as that $\sum_{j=2}^{k} j t_j \leq n$ because the total number of people whose debts were packed into zero-sum sets of size $\leq k$ cannot the total number of people, which is n. This expression is maximized at $t_j = 0$ for all j. This are two good explanations for this. The mathematical reason is that for each j, in the numerator we subtract $1 \cdot t_j$, and in the denominator we subtract $c \cdot t_j$ for $c < 1$. If our goal is to maximize this expression, then subtracting more from numerator than the denominator will

only make this expression smaller. Thus, it is best to set all the t_j's to 0 in order to maximize this fractional expression. The second reason is more intuitive. Our approximation algorithm finds the best packing of zero-sum sets of size $\leq k$. The worst situation that can happen is if our algorithm finds no sets at all because all the zero-sum sets of X are of size $> k$. This would imply that our approximation algorithm would require $n-1$ transactions to settle the debt, which is the maximum possible value for the numerator. Because of these two reasons, we have that the maximum occurs at $t_j = 0$ for all j, and the above is then bounded by $\frac{k+1}{k}$ as desired. □

3.3.2 $k = 2$

For $k = 2$, the set packing problem can be solved exactly in polynomial time via a maximal matching problem. We can create a graph where each element is represented by a vertex and each of the sets of size 2 is an edge connecting its two element's vertices. Then, finding the maximal set packing of these size 2 sets is equivalent to finding the maximal matching on this graph, which can be solved in polynomial time [3]. Therefore, by theorem 3.3.1 we get a simple $\frac{3}{2}$ algorithm for the minimum number of transactions.

3.3.3 $k = 3$

For $k \geq 3$, we have that the k-set packing problem is NP-complete [1] and so we don't expect to be able to find any polynomial time algorithms that can solve it exactly. Recall that there exists an existing polynomial time algorithm which can approximate the k-set packing problem to a factor of $\frac{k+2}{3}$ [2]. So one thing we could try is to simply plug in $\alpha = \frac{5}{3}$ into equation 3.8 and then use that as a bound for the numerator of equation 3.9. Concretely for $k = 3$, we would be trying to maximize the following expression:

$$\frac{n - s_2 - s_3}{\frac{3}{4}n - \frac{1}{2}t_2 - \frac{1}{4}t_3} \leq \frac{n - \frac{3}{5}t_2 - \frac{3}{5}t_3}{\frac{3}{4}n - \frac{1}{2}t_2 - \frac{1}{4}t_3} \tag{3.10}$$

We have the constraints $t_2, t_3 \geq 0$ and $2t_2 + 3t_3 \leq n$. Where does the maximum of the above expression occur? Using Mathematica, we find that the maximum occurs when $t_2 = \frac{n}{2}$ and $t_3 = 0$ in which case the above expression is equal to $\frac{7}{5}$. Therefore, if we use algorithm 2 with the $\frac{5}{3}$ approximation to the 3-set packing problem, we get a 1.4 approximation. This is not too bad, but can we do better?

It turns out that we can achieve a $\frac{4}{3}$ approximation ratio with a slight modification of our algorithm 2. Note that it is clear we can't do better than $\frac{4}{3}$ because that's the approximation ratio we get when all the t_i's are set to 0. The modified algorithm is as follows:

Algorithm 3 Modifies Algorithm 2 by taking the best out of many iterations

Suppose we are given as input the debts $X = \{x_1, x_2 \dots x_n\}$ as well as a constant k^*. We run algorithm 2 on X with $k = 2, \dots k^*$. When $k = 2$, we use the exact $2-$set packing algorithm. When $k \geq 3$, we use the $k-$set packing approximation with ratio $\frac{k+2}{3}$, Thus, define

$$\alpha_k = \begin{cases} 1 & k = 2 \\ \frac{k+2}{3} & k \geq 3 \end{cases}$$

Take the value of $k \in \{2, 3, \dots, k^*\}$ that gives the minimum number of transactions from running algorithm 2 on X with that value of k with an α_k approximation algorithm for the k-set packing problem. Equivalently, we took the $k \in \{2, 3, \dots, k^*\}$ that produced the largest number of disjoint zero-sum sets of size $\leq k$. Let s_{ij} be the number of sized j sets we took after running algorithm 2 for $k = i$. Let $s_2^*, s_3^*, \dots, s_{k^*}^*$ be the number of zero-sum sets of size $2, 3, \dots, k^*$ in this best solution from algorithm 2.

Lemma 3.3.3.

$$\sum_{j=2}^{k^*} s_j^* \geq \frac{1}{\alpha_k} \sum_{j=1}^{k} s_{kj}$$

for all $k \in \{2, 3, \dots, k^*\}$.

Proof. This equation is true by the way we picked s_j^*. s_{kj} are the number of size j zero-sum sets that we packed when we run algorithm 2 for k. We chose the s_i^*'s to be the number of size i zero-sum sets that were packed in the k from $\{2, 3, \dots k^*\}$ that gave the best total number of sets packed. So, in particular, the total number of sets packed for s^* must be greater than the total number of sets packed returned by running algorithm 2 on various values of k. Each value of k gave us an α_k approximation, so that's where the $\frac{1}{\alpha_k}$ comes from. □

Remark: One might wonder why is it that algorithm 3 does better than algorithm 2. The reason is that as k gets large, the approximation ratio decays. If we have an input $X = \{x_1, x_2, \dots, x_n\}$ where we can find $n/2$ disjoint zero-sum sets of size 2, that would certainly be the optimal solution. If we ran a 2-set packing algorithm, we would find this solution exactly. However, what if we ran a 3 or 4-set packing approximation here? We would only catch $\frac{3}{5} \cdot \frac{n}{2}$ or $\frac{3}{6} \cdot \frac{n}{2}$ of these size 2 sets. Therefore, we need to combine the 2 and 3 set packing approximations in order to get a better solution than just using the 3 approximation. Likewise, that is why for every value of k^*, we apply algorithm 2 for all $2 \leq k \leq k^*$.

For $k^* = 3$, we get the following constraints by lemma 3.3.3.

$$s_2^* + s_3^* \geq s_{22} \geq t_2$$
$$s_2^* + s_3^* \geq s_{32} + s_{33} \geq \frac{3}{5}(t_2 + t_3)$$

This gives rise to the following maximization problem based on equation 3.9:

$$\text{maximize } \frac{n - s_2^* - s_3^*}{\frac{3}{4}n - \frac{1}{2}t_2 - \frac{1}{4}t_3} \text{ subject to:}$$

$$s_2^* + s_3^* \geq t_2 \tag{3.11}$$

$$s_2^* + s_3^* \geq \frac{3}{5}(t_2 + t_3) \tag{3.12}$$

$$2t_2 + 3t_3 \leq n$$

$$2s_2^* + 3s_3^* \leq n$$

$$s_2^*, s_3^*, t_2, t_3 \geq 0$$

We notice that this maximization problem is really the same as the one we had above in equation 3.10, except instead of bounding it by another expression, we've added new variables (the s_i^*'s). The constraints 3.11 and 3.12 help us maintain that whatever s_i^* we choose is feasible in terms of the α_k approximations we used.

3.4 $k^* = 4$

For $k = 4$, it turns out that we can get a $\frac{13}{10}$ approximation to the minimum number of transactions by continuing to use this $\frac{k+2}{3}$ approximation to the k-set packing problem. We would get the following constraints:

$$s_2^* + s_3^* + s_4^* \geq t_2$$

$$s_2^* + s_3^* + s_4^* \geq \frac{3}{5}(t_2 + t_3)$$

$$s_2^* + s_3^* + s_4^* \geq \frac{1}{2}(t_2 + t_3 + t_4)$$

This gives us the following optimization problem:

$$\text{maximize } \frac{n - s_2^* - s_3^* - s_4^*}{\frac{4}{5} - \frac{3}{5}t_2 - \frac{2}{5}t_3 - \frac{1}{5}t_4} \text{ subject to:}$$

$$s_2^* + s_3^* + s_4^* \geq t_2$$

$$s_2^* + s_3^* + s_4^* \geq \frac{3}{5}(t_2 + t_3)$$

$$s_2^* + s_3^* + s_4^* \geq \frac{1}{2}(t_2 + t_3 + t_4)$$

$$2t_2 + 3t_3 + 4t_4 \leq n$$

$$2s_2^* + 3s_3^* + 4s_4^* \leq n$$

$$t_2, t_3, t_3, s_2^*, s_3^*, s_4^* \geq 0$$

The maximum of the above occurs at $t_2 = \frac{3}{16}n, t_3 = \frac{2}{16}n, t_4 = \frac{1}{16}n$ and $s_2^* = \frac{3}{16}n, s_3^* = 0, s_4^* = 0$. The maximum value comes out to be $\frac{13}{10}$ and so using algorithm 3 for $k^* = 4$ gives us a 1.3 approximation.

3.4.1 Conclusion

We started this section by presenting algorithm 2, which is a way for us to convert approximation algorithms for the k-set packing problem to an approximation for the minimum number of transactions in the debt settling problem. Algorithm 2 worked well for $k = 2$ giving us a $\frac{3}{2}$ approximation,

while choosing $k = 3$ gave us a $\frac{7}{5}$ approximation. At this point, we realized that we could get a better approximation by using algorithm 3 which is an enhancement over 2 because for $k^* = 3$, it takes the best out of algorithm 2 for $k = 2$ and $k = 3$. Seeing how algorithm 3 did better, we now moved on to trying $k^* = 4$ and that gave us an even better $\frac{13}{10}$ approximation

It turns out that if we had simply used algorithm 2 with $k = 4$ instead of algorithm 3 with $k^* = 4$, we would have gotten a $\frac{3}{2}$ approximation, which is what we got when we used $k = 2$ in algorithm 2. This is not surprising because algorithm 2 will not improve when you increase k. The reason is that the k-set packing approximation ratio decays as k increases. We already saw an example showing why this is true, but to retierate, imagine we ran algorithm 2 with $k = 19$. Then, our $k-$set packing approximation ratio would be 7, so we would catch 1/7 of the best packing with sets of size 19 or less. But what the optimal packing actually only uses zero-sum sets of size 2? Then we would catch only 1/7 of them. Yet we have a deterministic algorithm that when run on all the zero-sum sets of size 2 will find this optimal solution! Hence, algorithm 2 needs to be modified in order to be able to handle larger k's.

Chapter 4

Complexity

4.1 Overview

We have already seen from chapter 2 that the zero-sum set packing problem is NP-complete. The main reason for this NP-completeness is that it may take exponential time for us to even find a zero-sum subset at all. You are only given as input X, the underlying universe of numbers, and you have to compute the zero-sum subsets of X. This automatically makes the zero-sum set packing problem hard because finding any of the zero-sum subsets is exactly the subset sum problem, which is NP-complete.

But what if those zero-sum subsets were given you to you as part of the input? We would just be solving a set packing problem on those subsets. However, these subsets are not just arbitrary subsets: they are zero-sum and have the properties described at the end of chapter 2. We know that set packing is NP-complete and admits no constant factor approximation [4], but is this still true even when the sets we're allowed to choose are zero-sum and thus have special properties?.

We will also explore how the $k-$set packing problem is affected when all the given sets are zero-sum. The $k-$set packing problem is the variation of the set packing problem where all the given subsets are of size $\leq k$. First, let's define more formally what it means for our input to contain both X and the zero-sum subsets of X.

Definition 4.1.1. A set of integers S with $\sum_{s \in S} s = 0$ is called **primitive** if for all proper non-empty subsets T of S, we have $\sum_{t \in T} t \neq 0$.

Lemma 4.1.1. *Any optimal solution to the zero-sum set packing problem will only use primitive zero-sum subsets.*

Proof. Let $X = \{x_1, x_2 \dots x_n\}$ with $\sum_{i=1}^{n} x_i =$ be the input to the zero-sum set packing problem, and suppose $S = \{S_1, S_2 \dots S_t\}$ be our optimal solution. This means that the S_i's are all zero-sum subsets of X, disjoint, and t is as large as possible. The lemma claims that each of the S_i must be primitive. Without loss of generality, assume that S_1 is not primitive. Then, by definition 4.1.1, there exists some $T \subset S_1$ such that T is zero sum, and $T, S_1 - T \neq \emptyset$. But the subset difference property of zero-sum sets, we know that $S_1 - T$ is also zero-sum. Therefore, $S' = \{T, S_1 - T, S_2, S_3 \dots S_t\}$ is a collection of $t + 1$ zero-sum sets that are all mutually disjoint. This is a contradiction that S was the optimal packing to begin with because we have found another solution S' with $|S'| > |S|$. Therefore, it must be true that all the S_i's are primitive in the optimal solution. □

Definition 4.1.2. In the **k Zero-Sum Set Packing** problem, you are given as input $X = \{x_1, x_2 \dots x_n\}$ and

$$S = \{Y : Y \subset X, |Y| \leq k, Y \text{ is primitive zero-sum set } \}$$

Note that we guarantee S contains *all* primitive zero-sum subsets of X. The goal of the k-zero-sum set packing problem is to find the largest $T \subset S$ such that the elements of T are mutually disjoint.

Remark: Lemma 4.1.1 tells us that is is equivalent to pass in S as the set of all zero-sum sets of size $\leq k$ or pass in S as the set of all primitive zero-sum sets of size $\leq k$.

We are slightly moving away from the debt settling problem because we are not requiring that $\sum_{i=1}^{n} x_i = 0$. We showed in chapter 2 that the debt settling problem and the zero-sum set packing problem reduce to one another. In the zero-sum set packing problem (without the k in front), we are guaranteed that the sum of the x_i's in our input is 0. In the $k-$zero-sum set packing problem, we are given all the primitive zero-sum subsets, so if $\sum_{i=1}^{n} x_i = 0$, then X would appear in S only if X did not have any other zero-sum sets.

Definition 4.1.3. The **n Zero-Sum Set Packing** problem is a case of the k-zero-sum set packing problem where S contains all primitive zero-sum subsets of X.

This fits definition 4.1.2 because if $k = n$, then we have that S contains any sized primitive zero-sum subsets of X. The next theorem shows that the n-zero-sum set packing problem can be reduced to the zero-sum set packing problem (where we are not given S as part of the input). In particular, this shows that the n-zero-sum set packing problem can further reduced to the debt settling problem.

Theorem 4.1.2. *The n-zero-sum set packing problem reduces to the zero-sum set packing problem*

Proof. Recall that there are two differences between these problems: in the n-zero-sum set packing problem, we are given X which does not need to sum to 0, and S which contains all the zero-sum subsets of X. In the regular zero-sum set packing problem, we are only given X', which is a zero-sum set.

Given an instance of the n-zero set sum packing problem with $X = \{x_1, x_2, \dots, x_n\}$ and $S = \{Y : Y \subset X, Y \text{ is a primitive zero-sum set}\}$, we show how we can convert this into an instance of the zero-sum set packing problem (where we require that X be a zero-sum set). Our reduction is very simple, let $x' = \sum_{i=1}^{n} x_i$. We will assume that $x' \neq 0$ or else the two problems are already equivalent. Add $-x'$ to X to get X'. Now, X' is a zero-sum set and therefore it is a valid input to the zero sum set packing problem.

The claim is that if X has a zero-sum set packing of size k then X' has a zero-sum set packing of size $k+1$. Suppose $Y_1, Y_2, \dots, Y_k$ is our zero-sum set packing of X. Let $Y^* = \{x'\} \cup (X - Y_1 - Y_2 - \dots - Y_k)$, and we notice that Y^* is also a zero-sum set because of the way x' was chosen. Then, it is clear that $Y_1, Y_2, \dots, Y_k, Y^*$ is a zero-sum set packing of X' of size $k + 1$. This shows that we can find the optimal n-zero-sum set packing problem using an optimal zero-sum set packing algorithm as a black box in the following way:

Take X and construct X' as defined above. Pass X' into the zero-sum set packing to get the optimal sets $X_1, X_2, \dots, X_k$, so k is as large is possible. This implies that the X_i's are primitive zero-sum sets because of lemma 4.1.1. Without loss of generality, suppose $x' \in X_1$. This means that $X_i \in S$

for all $i \geq 2$, but $X_1 \notin S$ because X_1 includes the extra element x' that we added to X. This means that $X_2, X_3, X_4, \ldots X_k$, which are all in S is a valid n-zero-sum packing of X. This set of $k-1$ subsets must be the optimal packing of X because we proved that a zero-sum packing of size $k-1$ for X implies a zero-sum set packing of size k for X', but we know that X' has a maximal zero-sum set packing of of size k.

Therefore, we get that we can use a black box for the zero-sum set packing problem to solve the n-zero-sum set packing. □

The main result of this chapter is to show that the k zero-sum set packing problem is hard for $k \geq 3$, including the n zero-sum set packing problem. We also show that although intuitively there might be hope that the n-zero-sum set packing problem admits better approximations than the general set packing problem, it turns out that there is a bi-directional approximation preserving reduction between the set packing problem and the n-zero sum set packing problem. This will be an interesting result because it is clear that the n-zero sum set packing problem can be reduced to the general set packing problem, but it is surprising that any instance of the general set packing problem can be converted to an instance of the n-zero-sum set packing problem. This bidirectional reduction will also show that it is impossible to approximate the n-zero-sum set packing problem better than the general set packing problem.

The $k = 1$ case is trivial and when $k = 2$, the $k-$zero sum set packing problem can be solved in polynomial time. Finding the optimal packing of X with sets of size 2 can be done greedily, by taking each element $x \in X$ and checking if $-x \in X$. If so, remove both x and $-x$ from X and recurse. This is correct because the only zero sum sets of size 2 are of the form $\{x, -x\}$ and therefore if x is going to be packed, it has to be with $-x$ so we can add it greedily to our solution. The more interesting cases are when $k > 2$.

4.2 $k \geq 4$

We will show that for $k \geq 4$, the k-Zero Sum Set Packing problem is NP-complete via a reduction from maximal independent set. We require that $k \geq 4$ because we will reduce the k-zero sum set packing problem from maximal independent set on a graph with degree $k-1$. This is known to be NP-complete for $k - 1 \geq 3$ and hence we require $k \geq 4$.

Our goal is to take an instance of independent set and generate a set of numbers and primitive zero-sum subsets of those numbers of size $\leq k$ such that solving the zero-sum set packing problem on that input leads to a solution to the original independent set instance. The most informative way to show this reduction is to start with an example.

4.2.1 Example

Suppose you were given the following graph G and asked to find the maximal independent set. Notice that there is a positive weight on each edge, which will be used later.
For each vertex v, we create a set S_v which contains $w(e)$ for each edge e incident to v. It also

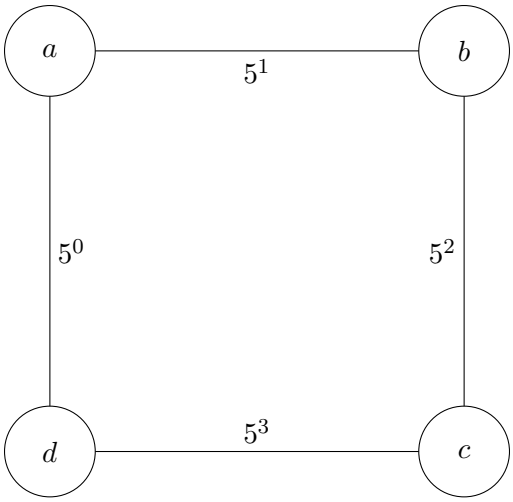


Figure 7: Example of Independent Set to Zero Sum Set Packing

contains an negative element, which is the sum of all those $w(e)$'s.

$$\begin{aligned}
S_a &= \{1, 5, -6\} \\
S_b &= \{5, 25, -30\} \\
S_c &= \{25, 125, -150\} \\
S_d &= \{1, 125, -126\} \\
X &= S_a \cup S_b \cup S_c \cup S_d = \{1, 5, 25, 125, -6, -30, -126, -150\}
\end{aligned}$$

We have that X along with $S = \{S_a, S_b, S_c, S_d\}$ is a valid instance of the 3-zero-sum set packing problem because S contains all the sized 3 or less primitive subsets of X. Any set packing of S will represent an independent set in the graph G.

For example, we see that a, c is valid independent set of G, and hence S_a and S_c is a valid packing. Hence, we have turned the question of finding an independent set on a graph with degree bounded by 2 to a $3-$zero-sum set packing problem.

4.2.2 Generalization

Theorem 4.2.1. *The k Zero-Sum Set Packing Problem is NP-complete for $k \geq 4$.*

Proof. Given an instance of the maximal independent set on a graph $G = (V, E)$ where the degree of every vertex is at most $k - 1$, we will show how to convert this into an instance of the k zero-sum set packing problem. This implies that if the k zero-sum set packing problem had an efficient solution, then so would maximal independent set on bounded degree graphs, which is NP-complete for degree ≥ 3. In particular, this reduction preserves approximations, and thus it is impossible to approximate the k set packing problem better than the $k-1$ independent set problem.

Let $|V| = n$ and $|E| = m$ as usual. Suppose $E = \{e_0, e_1, e_2 \dots e_{m-1}\}$. We assign the weight of each edge to be $w(e_i) = (n+1)^i$. Let $\text{adj}(v)$ be the set of edges with one end point as v. For each vertex, we will also assign it a weight, but in the following way:

$$w(v) = -1 \cdot \sum_{e \in \text{adj}(v)} w(e)$$

Now, the claim is that the size of the maximum set packing of $X = \{w(v) : v \in V\} \cup \{w(e) : e \in E\}$ with zero-sum subsets of size $\leq k$ is exactly the size of the maximal independent set in G.

What are the primitive zero-sum subsets of X of size $\leq k$? By the way we constructed X, we definitely have the set S_v for each $v \in V$:

$$S_v = \{w(e_i) : e_i \in \text{adj}(v)\} \cup \{w(v)\}$$

The next two lemmas will show that these S_v's are indeed the only primitive subsets of X.

Lemma 4.2.2. *All of the S_v's are primitive zero-sum sets.*

Proof. By our construction of the weights, we have that each S_v contains only 1 negative element, namely $w(v)$. Every other element is equal to $(n+1)^i$ for some i. Therefore, there cannot exist a proper subset of S_v which sums to 0 because any such set would need to use the negative element $w(v)$, but if one does so then one would have to use all the remaining elements as well. □

Lemma 4.2.3. *The only primitive zero-sum subsets of X are the S_v's.*

Proof. This is an interesting lemma that shows the motivation behind choosing $(n+1)^i$ as the weights of the edges. Let S be an arbitrary **primitive** zero sum subset of X. Let's do case work depending on how many negative elements S has:

- *One.* If S has one negative element, then suppose that $w(v) \in S$ for some $v \in V$. This is true because the $w(v)$'s are the only negative numbers in W. By our construction of choosing the edge weights to be powers of $n+1$, there is a unique combination of the edges whose weights sum up to $w(v)$. This unique combination can be thought of as writing $w(v)$ in base $n+1$. Therefore, if the negative number $w(v)$ exists in X, then the only way it could have come about is from S_v, which contains $w(v)$ and the unique edge weights adding up to $w(v)$. Therefore, S must be equal to S_v.

- *Two.* Our goal is to show that if S contains 2 of the negative $w(v)$'s, then it must not be primitive. This is true because of the edge weight's base of $n+1$ that we chose. Suppose $w(u) \in S$ and $w(v) \in S$. If we wrote $-w(u)$ and $-w(v)$ in base $n+1$, they would look binary (i.e. be only 1's and 0's). When we add them together, the resulting number would have only 0's, 1's and 2's in base $n+1$. If this number has a 2 in it, then it would be impossible to find a zero-sum subset containing both $w(u)$ and $w(v)$ because the positive numbers are all of the form $(n+1)^i$ and there is only 1 each for every i. If the number does not contain a 2, then we must have had something where $-w(u)$ and $-w(v)$ do not overlap in which digits they have 1's in base $n+1$. Therefore, the $\text{adj}(u) \cap \text{adj}(v) = \emptyset$ and this implies S is not primitive because it can be split into $\{w(u)\} \cup \text{adj}(u)$ and $\{w(v)\} \cup \text{adj}(v)$.

- *Three or More.* We chose a base of $n+1$ because there are only n negative numbers, so even if you added them all together, you would not be able to form a number that is binary in base $n+1$ unless all n negative numbers had distinct digits in base $n+1$, in which case S would not be primitive.

This completes the proof that all the primitive non-zero subsets of X are the S_v's. This means that the optimal set packing over the S_v's is of the same size as the optimal set packing of X using any of its zero-sum sets. Note that each S_v has cardinality k if our given graph G has degree $k-1$. □

Using Lemma 4.2.3, we are pretty close to proving Theorem 4.2.1. We just need to show that G has a maximal independent set of size t if and only if W has a zero-sum set packing of size t. The following two lemmas are the forward and backward directions of the above statement.

Lemma 4.2.4. *G has an independent set of size $t \Rightarrow X$ has a zero-sum packing of size t.*

Proof. Let $v_1, v_2 \dots v_t$ be an independent set of G. Thus means that $\text{adj}(v_1) \dots \text{adj}(v_t)$ are all disjoint by the definition of an independent set. This means that $S_{v_1}, S_{v_2} \dots S_{v_t}$ all do not intersect because each S_v was constructed by adding $w(v)$ and $w(e)$ for $e \in \text{adj}(v)$. Therefore, this is a zero-sum packing of size t. □

Lemma 4.2.5. *X has a zero-sum set packing of size $t \Rightarrow G$ has an independent set of size t.*

Proof. We know that the only primitive zero-sum sets of W are the S_i's and therefore there must exist t such S_v that are disjoint. Without loss of generality, assume those sets are $S_{v_1} \dots S_{v_t}$. Then, $v_1, \dots v_t$ must be an independent set because if there was an edge e between v_i and v_j, then S_{v_i} and S_{v_j} would both contain the element $w(e)$ and hence not be disjoint anymore. But we are given the assumption that all the $S_{v_1}, \dots S_{v_t}$ are disjoint, so this cannot be the case. Hence, $v_1, \dots v_t$ is an independent set and thus we have shown that an independent set of size t exists. □

Combining together Lemma 4.2.4 and Lemma 4.2.5, we have proven Theorem 4.2.1 because we have shown that every independent set in G corresponds to a zero-sum packing of X of the same size and vice versa. □

4.3 $k = n$

Since $k = n$ is a case of $k \geq 4$, we already have by the previous section that the n-zero-sum set packing problem is NP-complete. Even though the sets in the n-zero-sum set packing problem have the unique structure of zero-sum sets, they do not make the problem easier. However, one might ask whether the n-zero-sum set packing problem is easier to approximate than the general set packing problem. For example, the general set packing problem admits no constant approximations [4], so one might wonder if that is true for the n-zero-sum set packing problem as well.

Theorem 4.3.1. *The n-zero-sum set packing problem is just as hard to approximate as the general set packing problem.*

Proof. We will show this by giving a approximating preserving reduction from the general set packing problem to the n-zero-sum set packing problem. This is interesting because the n-zero-sum set packing problem reduces to the general set packing problem in the canonical way, so we will see that the reduction also holds in the other direction. The reduction is done via the independent set problem.

Any instance of the set packing problem can be reduced to an instance of independent set in the following way: Given an instance of the set packing problem, $S = \{S_1, S_2 \dots S_n\}$, we create a graph with n nodes, $v_1, v_2 \dots v_n$. If $S_i \cap S_j \neq \emptyset$, then we add the undirected edge (v_i, v_j) in our graph. After doing this for all i, j pairs, the claim is that there is a 1-1 correspondence between a valid packing of S and an independent set of the graph. The proof of this is quite immediate because a valid set packing means that no two sets intersect, but two sets intersect if and only if there is an edge between their corresponding vertices in the graph.

In our proof of theorem 4.2.1, which claimed that the k zero-sum set packing problem is NP-complete for $k \geq 4$ and in particular $k = n$, we showed how to take an instance of maximal independent set, assign weights to the nodes and edges in order to create an instance of the n zero-sum set packing problem. The size of any solution does not change, meaning that a size t independent set corresponds to a zero-sum set packing of size t. Moreover, the size of the optimal packing does not change after performing this reduction. Therefore, this is an approximation preserving reduction from the general independent set problem to the n-zero-sum set packing problem. If we had an α approximation to the n-zero-sum set packing problem, we could use this transformation to get an α approximation for the general set packing problem. This implies that the n-zero-sum set packing problem is at least as hard to approximate as the general set packing problem. □

Note that we do not have the result that the k-zero-sum set packing problem is just as hard to approximate as the k-set packing problem because given an input of 3-set packing, if we transform that into an independent set problem using the method in the theorem above, the graph that we would be finding the maximum independent set over is not necessarily of degree 3. Therefore, the 3-set packing problem reduces to the n-zero-sum set packing problem, not to the 3-zero-sum set packing problem.

4.3.1 Example

Here is an example of the full reduction from regular set packing to n-zero-sum set packing. Let our input to the set packing problem be $S = \{\{a, b, c\}, \{d, e\}, \{b, c, d\}\}$. Note that if these subsets were zero-sum, then we would also need $\{a, e\} \in S$ because $\{a, e\} = (\{a, b, c\} \sqcup \{d, e\}) - \{b, c, d\}$, but that is not the case. This shows that it certainly does not have to be true that our input S follows the structure of zero-sum subsets, yet we will convert finding the maximal set packing of S to finding the maximal zero-sum set packing of something else.

First, we turn S into an instance of maximal independent set. This is done by making a node for each set and connecting two nodes if and only if their intersection is non-empty.

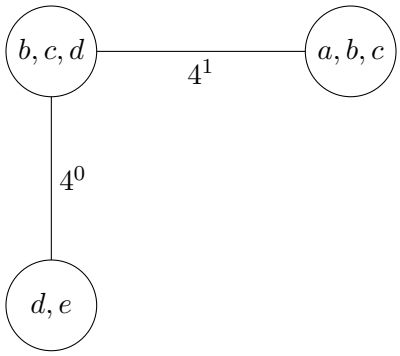


Figure 8: Example of Set Packing to Zero-sum Set Packing

Now, we convert this independent set instance into an instance of the n-zero-sum set packing problem by doing what we did in the proof of theorem 4.2.1. This results in the following sets and

universe X:

$$\begin{aligned} S_{d,e} &= \{1, -1\} \\ S_{b,c,d} &= \{1, 4, -5\} \\ S_{a,b,c} &= \{4, -4\} \\ X &= \{1, 4, -1, -4, -5\} \end{aligned}$$

Therefore, finding the maximum set packing of S is equivalent to finding the n zero-sum set packing of X with sets $S_{d,e}, S_{b,c,d}, S_{a,b,c}$. The overall idea of this reduction is to take an instance of set packing and create an instance of the n-zero-sum set packing problem with the same intersection relationships as the set packing instance.

4.4 $k = 3$

We have yet to show that the 3-zero-sum set packing problem is NP-complete. We will do this via a reduction from a lesser known NP-complete problem, the numerical 3-dimensional matching problem [5].

Definition 4.4.1. In the **Numerical 3-Dimensional Matching** problem, we are given as input $X = \{x_1, x_2, \ldots x_n\}, Y = \{y_1, y_2 \ldots y_n\}, Z = \{z_1, z_2, \ldots z_n\}$ positive integers such that the sum of all the x, y and z's is $3B$ for some positive integer B. The goal is to find a perfect 3-dimensional matching, meaning a partitioning of $X \cup Y \cup Z$ into n triples such that each triple contains exactly 1 element from each of X, Y and Z and the sum of the elements in each triple is B.

This problem has been shown to be NP-complete in the strong sense by Garey and Johnson **??**.

Theorem 4.4.1. *There is a polynomial time reduction from numerical 3-dimensional matching to 3-zero-sum set packing.*

Proof. Given an instance of the 3-D matching problem as defined above, we create the following sets:

$$\begin{aligned} X' &= \{x_1 + 3B, x_2 + 3B, \ldots, x_n + 3B\} \\ Y' &= \{y_1 + 4B, y_2 + 4B, \ldots, y_n + 4B\} \\ Z' &= \{z_1 - 8B, z_2 - 8B, \ldots, z_n - 8B\} \end{aligned}$$

Now, we can run a simple $O(n^3)$ algorithm to find all the zero-sum sets of size at most 3 and then filter for just the primitive ones. The following two lemmas demonstrate that there are no zero-sum sets of size 1 or 2.

Lemma 4.4.2. $0 \notin X' \cup Y' \cup Z'$.

Proof. All the x, y, z's started out as positive, so no element of X' or Y' can be 0. We know that the sum of all the x, y, z's is $3B$ and thus all the elements of Z' must strictly be negative. Therefore, there is no 0 in $X' \cup Y' \cup Z'$. □

Lemma 4.4.3. *If $a \in X' \cup Y' \cup Z'$ with $a \neq 0$, then $-a \notin X' \cup Y' \cup Z'$.*

Proof. We know that X' and Y' only have positive numbers, while Z' only has negative numbers. Suppose $(x_i + 3B) + (z_j - 8B) = 0$. This implies that $x_i + z_j = 5B$, which is impossible because the sum of all the x, y, z's is $3B$. Therefore, no two elements of X' and Z' are negatives of each other. The same argument can be used for Y', where if $(y_i + 4B) + (z_j - 8B) = 0$, then it would have to be true that $y_i + z_j = 4B$, which again is impossible. □

From the above two lemmas, we see that $X' \cup Y' \cup Z'$ has no zero-sum subsets of size 1 or 2. It only has zero-sum subsets of size 3, and it is clear that $x_i + y_j + z_k = B$ if and only if $(x_i + 3B) + (y_j + 4B) + (z_k - 8B) = 0$. Therefore, if we run our black box 3−zero-sum set packing problem on $X' \cup Y' \cup Z'$, we can know that any solution we get will only use zero-sum sets of size 3, and if the solution is of size n, then we can answer yes to the numerical 3-D matching problem. otherwise, we answer no to the numerical 3-D matching problem. □

It seems pretty simple to show that the 3-zero-sum set packing problem is NP-complete, so why did we do all that work to show that 4+ zero-sum set packing is NP-complete via independent sets? That reduction from independent set allows us to make a bidirection approximation preserving reduction between set packing and zero-sum set packing. However, there has yet to be discovered a reduction from the 3 zero-sum set packing problem to the numerical 3-D matching problem. Therefore, although this proof is simpler for showing that k zero-sum set packing is NP-complete for $k \geq 3$, it is not as valuable as the proof given above in terms of making claims about the approximate hardness of the k zero-sum set packing problem.

In this chapter, we looked at the complexity of the zero-sum set packing problem when we remove the bottleneck of having to find zero-sum sets. Even with all the primitive zero-sum sets as part of the input, we discovered that the k-zero-sum set packing problem was still hard for $k \geq 3$. In fact, at least for $k = n$, it is just as hard to approximate as the general set packing problem. It is still an open question as to whether the k-zero-sum set packing problem can be approximated or even solved faster than the k-set packing problem for a constant k.

Chapter 5

Settling Debt over a Graph

5.1 Introduction

An interesting variation of the debt settling problem is the debt settling problem over an incomplete graph. In this variation, we assume that not all of the n people are friends with each other and hence transactions between certain people cannot happen. This is a pretty realistic assumption because often times you will travel with people whom you do not have on Venmo, Paypal...etc. When it is time to settle the debt, only transactions among friends can be made. Given this restriction, the question now is to find the smallest set of these valid transactions that will settle all the debt.

This problem is especially relevant if we think of the agents as businesses instead of individuals. Businesses often owe each other money in exchange for goods and services. It would be great if instead of a business having to pay 10 other businesses separately for 10 purchases made that they only have to pay once. This would cut down on a lot of bureaucratic hassle of sending money from one business to another. However, payments between businesses are a bit more complicated than peer-to-peer payment systems. Only certain pairs of companies have established a mutual payment system, so they can easily pay each other, but many pairs of companies are unable to easily send money to each other. Therefore, it is a practical problem to find the minimum number of transactions necessary to settle the debt over an incomplete graph. We will, however, make the assumption that the graph is connected or else it wouldn't be possible to settle the debt at all.

Definition 5.1.1. The **Debt Settling Problem over a Graph** is a variation of the debt settling problem where you are given a connected graph $G = (V, E)$ with $V = \{v_1, v_2 \dots v_n\}$. For each vertex v_i, let's define its weight $w(v_i) = x_i$ to be the debt of the ith agent, with the overall property that $\sum_{i=1}^{n} x_i = 0$. The goal is to find the smallest set of transactions $\{(s_j, r_j, a_j)\}_{j=1}^{t}$ that settles the debt (see definition 1.2.2) and has the property that $(v_{s_j}, v_{r_j}) \in E$ for all j.

Remark: Note that if E is the complete graph on n vertices, then the debt settling problem over a graph is equivalent to the debt settling problem defined in definition 1.2.3.

What is an upper bound on the number of transactions needed to settle the debt over a graph? We saw that when our graph is complete, you can always settle the debt among n agents in $n-1$ transactions by having everyone pay or be paid by one person. This can no longer happen for a general connected graph because there may not be 1 person who can pay everyone. It turns out that it is still possible to settle the debt in $n-1$ transactions.

5.2 The Natural Solution

The easiest way to show how the natural solution is generalized to payments over a graph is through an example. Consider the following graph where the nodes represent people, the number inside the node is their debt, and the edges represent possible payments.

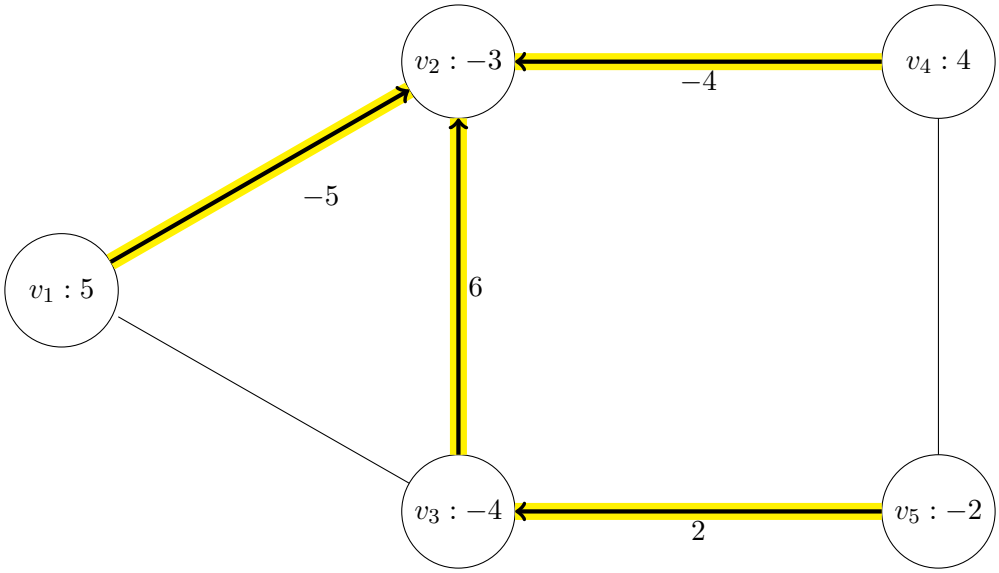


Figure 9: The Natural Solution for Graphs

We notice that in the above example, if all transactions were allowed, then we would simply have v_3 pay v_4, and then v_2 and v_5 can both pay v_1. That would take only 3 transactions. However, this is not possible because not all of those edges exist in the graph. It turns out that the minimum number of transactions needed is 4.

In a connected graph with n nodes, it is always possible to settle the debt in $n-1$ transactions. Let $T \subset E$ be a spanning tree of G, like the one highlighted in yellow above. Our algorithm for finding a set of $n-1$ transactions that settle the debt will be: Start by finding any spanning tree of G, call it T. Define an arbitrary node as the root and define parent-child relationships on T. For each leaf node of T, create a transactions between the leaf node and its parent. Remove the leaf node, update the parent nodes' balance and then recurse until all nodes have been removed from the graph and we have our set of transactions. This is clearly a polynomial time algorithm and some pseudocode is given below:

Algorithm 4 Natural Solution on a Graph

```
1: procedure FindTransactions(V, T)
2:     V ← The vertices of the payment graph with V.w being the weight function assigning each
   vertex in V a debt.
3:     T ← Any fixed spanning tree of G with some fixed parent-child relations.
4:     return ← a set of transactions satisfying definition 5.1.1
5:
6:     v_c ← any leaf node of T
7:     v_p ← the parent node of v_p
8:     if v_p = Null then
9:         assert(V.w(v_c) == 0); return [ ]
10:    transaction ← (v_c, v_p, V.w(v_c))
11:
12:    V.w(v_p) ← V.w(v_p) + V.w(v_c)
13:    V ← V − {v_c}
14:
15:    solution ← findtransactions(V, T)
16:    return solutions.append(transaction)
```

Theorem 5.2.1. *Algorithm 4 correctly settles the debt over the graph for all the agents in $n-1$ transactions.*

Proof. This is a simple proof by induction. Staring with a child node v_c with debt $V.w(v_c)$, the transaction $(v_c, v_p, V.w(v_c)$ settles the debt for v_c. For example, if $w.V(v_c) = -2$, then v_c is owed 2 dollars, and so having the transaction $(v_c, v_p, -2)$ is equivalent to $(v_p, v_c, 2$, or that the parent pays the child \$2. This settles the debt of the child and so he/she can be removed from the graph. As a result of this payment, the parent's debt needs to be changed. In the example, after paying the child 2, the parents' debt decreases by 2. If we inductively assume that our algorithm is correct on the rest of the graph after removing this child node and modifying the parent's balance, then our algorithm is correct for the entire graph including the children. □

Remark: Algorithm 4 always returns a set of $n-1$ transactions (one for each edge of the spanning tree), but some of the amounts for the transaction may be 0. In that case, we can remove the transaction.

In the next two sections, we will look at two types of graphs: trees and grids of fixed width. We will show that when G is a tree, there is an efficient polynomial time algorithm for finding the optimal set of transactions. We will also show that when G is a fixed-height grid, G admits a psuedo-polynomial algorithm for finding the optimal set of transactions.

5.3 G is a Tree

When G itself is a tree, there is not much flexibility in what transactions can happen for the debt to be settled. We know that any tree T must contain at least 1 vertex with degree 1. For that one vertex, there is no choice as to how its debt will be settled. Thus, our algorithm will find these nodes with degree 1 and settle their debt. The correctness of our algorithm in finding the minimum number of transactions will rely on the fact that the optimal solution must make the same choices as we do.

Algorithm 5 Finding the optimal sequence of payments on a tree

Suppose our graph is T, where T is a tree. There must exist a node $t \in T$ such that t is has only 1 adjacent neighbor, call it s. If $w(t) = 0$, then simply remove t and recurse. Otherwise, in order to settle t's debt, which is $w(t)$, there must be a transaction between t and s, assuming that $w(t) = 0$. We add the transaction $(s, t, w(t))$ to our list of transactions. Once we have made this transaction, s's balance has now increased by $w(t)$. Therefore, we increase $w(s)$ by $w(t)$. Remove t from T and recursively apply this algorithm to the $T - \{t\}$, which is still a tree.

Theorem 5.3.1. *Algorithm 5 correctly finds the minimum number of transactions needed.*

Proof. We will prove that every time we add the transaction $(s, t, w(t))$ into our solution, it must be part of optimal solution as well. In this sense, our greedy algorithm stays on track with the optimal solution because every choice we make is reflected in the optimal solution.

Why is it the case that at each iteration of our algorithm, the optimal solution must also contain exactly one copy of $(s, t, w(t))$ if $w(t) \neq 0$? The reason is that t only has 1 neighbor, which is s. If t's debt is settled, it has to be because of transactions with s. It is clear that in the optimal solution, there can only be 1 transaction between s and t because multiple transactions between the pair can be aggregated into a single transaction. The amount for this transaction must be $w(t)$ if t is to have all his or her debt settled. Therefore, $(s, t, w(t))$ must appear in the list of transactions.

Since the order in which the transactions are made do not matter for the debt settling problem, we can assume that $(s, t, w(t))$ is the first transaction to take place. Now, we're left with a situation where s's balance is updated and we're trying to find the best way to settle the debt. Inductively, algorithm 5 will find the optimal sequence of transactions there as well. □

Therefore, we have shown that the debt settling problem is easy when the underlying graph is a tree. Next, we will show that when the underlying graph is a grid, we can

5.4 G is a Fixed-height Grid

A fixed height grid is a grid of n elements where the height of the grid is bounded by c for some constant c. Figure 10 shows an example of a grid bounded in height by 3.

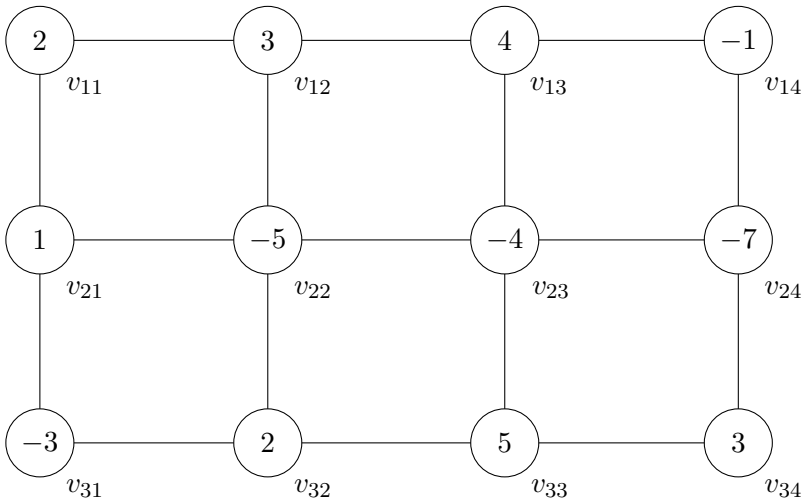


Figure 10: Our Example for Grid G with fixed height

The optimal sequence of payments is represented by the following forest of highlighted edges. Within each forest, the sum of the debts of all the people are 0, so we can use the natural way on a graph to settle their debts. For example, we might have v_{11} pay v_{12} \$2, and then have v_{12} pays v_{22} \$5.

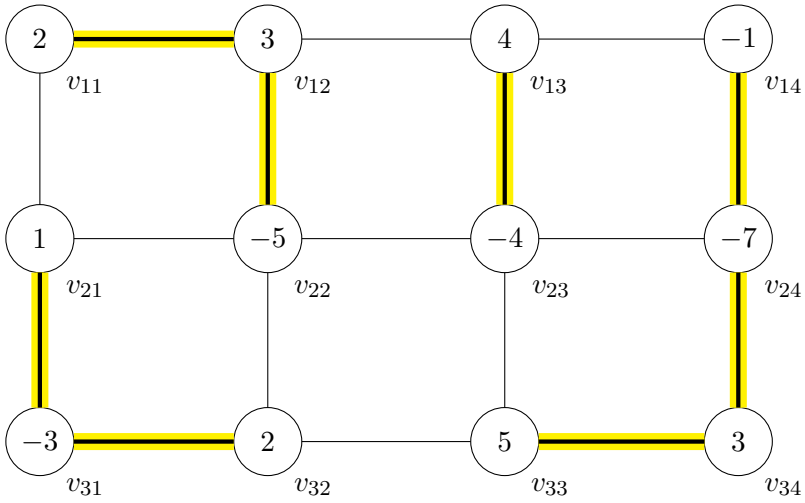


Figure 10: Our Example for Grid G with fixed height

The goal of this section is to present a pseudo-polynomial time dynamic programming algorithm for finding the minimum sequence of transactions for the debt settling problem over a graph when the graph has is a fixed-height grid. The main idea of the dynamic programming algorithm is that we will start with the left-most column of people, v_{11}, v_{21} and v_{31}. We will consider all the ways in which they can make payments to settle their debt. The following figure gives a few examples of the ways in which these 4 people can settle their debt, either by paying each other or by paying the people immediately to their right.

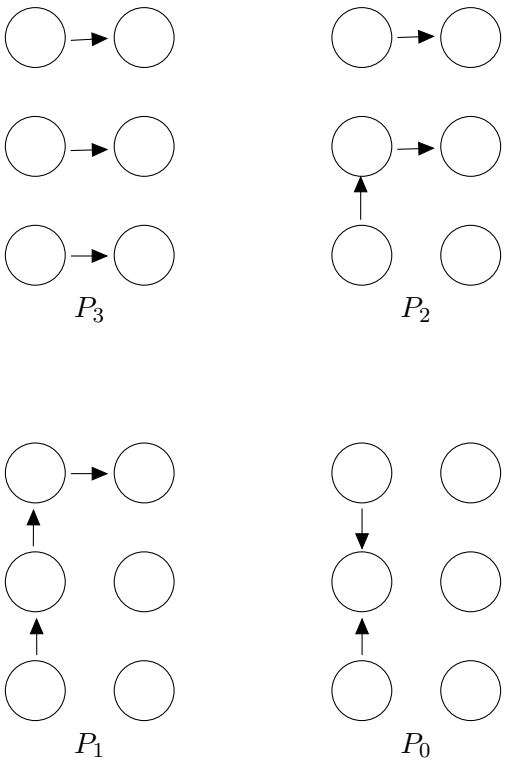


Figure 11: Ways in which payments can happen

Note that the figure shows ways in which all $0, 1, 2$ or 3 people make payments to the column of people to the right. For simplicity, it does not show all possible ways for that to happen, but just 1 example of each. For example, in the bottom left graph, it is also totally valid for the middle person to make a payment to the right and the top and bottom people pay the middle person. Nevertheless, we can see that for a fixed c, there are only a constant number of ways for the three people in the left-most column to settle their debts. Also, note that while the arrows seem to imply money being transferred in that direction, our definition of transaction permits negative amounts to be transferred, which would effectively make the arrow point in the other direction.

As per the definition of the debt settling problem over a graph, suppose we are given a grid graph $G = (V, E)$ of height c and width n for some constant c. Let the vertex on the ith row and jth column be labeled v_{ij}, and suppose that $w(v_{ij})$ gives us the debt of person v_{ij}. This grid has c rows

and $\frac{n}{c}$ columns, and we know that

$$\sum_{i=1}^{c}\sum_{j=1}^{\frac{n}{c}} w(v_{ij}) = 0$$

in order for these to be valid debts. Also, let's define

$$M = \sum_{i=1}^{c}\sum_{\frac{n}{c}}^{c} |w(v_{ij})|$$

Definition 5.4.1. Over a grid graph, a transaction $(v_{ij}, v_{i,j+1}, b)$ is considered a **Right** transaction because b dollars are sent from an agent to their immediate right neighbor in the grid. Similarly, a transaction of the form $(v_{ij}, v_{i+1,j}, b)$ are a **Down** transaction, while a transaction of the form $(v_{ij}, v_{i-1,j}, b)$ is called an **Up** transaction.

Definition 5.4.2. A **Column Payment Pattern** is a $c-$tuple $P = (d_1, d_2, d_3 \dots d_c)$ where $d_i \in \{U, D, R, N\}$, which stand for the directions Up, Down, Right and Nothing. It represents how the people in the left-most column of a grid will resolve their debt, from top to bottom. For example, in Figure 11, $P_3 = (R, R, R), P_2 = (R, R, U), P_3 = (R, U, U), P_4 = (D, N, U)$.

Let's exclude the patterns where there is a D at index i and U at index $i + 1$ of the pattern. That would mean that the ith agent pays the $i + 1$st agent in the column and the $i + 1$st agent pays the ith agent in the column. We know that no optimal solution would every make two people have more than 1 transaction between them.

Definition 5.4.3. Let $x_1, x_2, \dots, x_c$ be the debts of the left-most column's agents from top to bottom. We say that a column payment patter $P = \{d_1, d_2, \dots, d_c\}$ is **valid** with respect to $x_1, x_2, \dots x_c$ if it is possible to settle all of the left-most column agents' debts by using transactions according to P.

Example 5.4.1. In the following example, for the same column payment pattern, the left is valid while the right is not. If these were the only transactions allowed for the left-most column's agents, then in the right situation, the debts of bottom two agents would not be resolved.

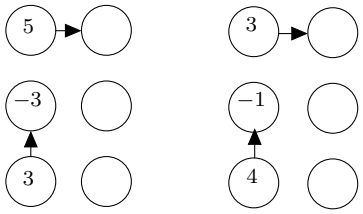


Figure 12: Valid and invalid column payment patterns

When we apply a column payment pattern to a the k column, in most patterns, we will need to update the debts of the people in the $k + 1$st column because there will be some collection of Right transactions.

Example 5.4.2. Suppose we used column payment pattern P_2 on the left-most column of the example from Figure 10. The debts of the agents the second column from the left will be $w(v_{12}) = 5, w(v_{22}) = -7$ and $w(v_{32}) = 2$ This is because column payment pattern P_2 makes agent v_{11} pay

agent v_{12}, which absolves v_{11} of his debt. In addition, v_{31} pays v_{21} -3 dollars, and then v_{21} pays v_{22} -2 dollars because v_{21} had 1 dollar of debt, but paid v_{31} a total of 3 dollars, and so v_{21} is now owed 2 dollars.

Lemma 5.4.1. *A valid column payment pattern determines the amounts that will be sent for each transaction.*

Proof. For any column payment pattern, every node in the left-most column has at most 1 arrow leaving it. That means that all of that agent's debt must be solved via that one arrow leaving it. Therefore, the amount sent in that transaction is fixed. As a result, each of the up to c transaction in a column payment pattern has only 1 possible amount associated with it if we want to resolve everyone's debt in the left-most column. Note that this amount can be negative, but we assumed throughout this entire paper that negative amounts can be sent and are equivalent to a transaction in the other direction. We are trying to minimize the total number of transactions, not the number of transactions any individual participates in, so this flexibility in the definition of a transaction does not hurt us. □

Finally, here is how our Dynamic Programming algorithm for settling debt on a c by $\frac{n}{c}$ grid will work:

Algorithm 6 DP algorithm for settling debt on a fixed height grid

Suppose we have a grid graph G with vertices labeled v_{ij} for the ith row jth column, and $w(v_{ij})$, their debts. Our recursive function $f(k, m_1, m_2, \ldots, m_c)$ represents the best way to settle the debt for all the agents whose column number is $\geq k$ with the additional property that everyone in the kth column's debts are overwritten by $w(v_{ik}) = m_i$ for $i = 1, 2, \ldots, c$. The debt of everyone with column number $> k$ still has their original debt, $w(v_{ij})$, but the people in the kth column have debt $w_1, w_2, \ldots, w_c$ instead.

Base Case: The base case is when $k = \frac{n}{c}$. In that case, we look at all valid column payment patterns that would settle the debt for this last (right-most) column. Because this is the right-most column, we do not allow any of the transactions to be Right transactions when choosing which column payment patterns are valid to resolve this column's debt. We choose the column payment pattern that uses the minimum number of transactions.

Recursive Case: For this column k, we iterate through all possible valid column payment patterns and apply each of them separately to this kth column. Lemma 5.4.1 tell us that any particular column payment pattern we choose determines a set of transactions that have to happen involving the agents in the kth column. For each valid payment pattern P, if it uses any Right transactions, then we will have to modify the debt of an agent in the $k+1$st column, just like we saw in example 5.4.2. Let $m'_j = P(v_{j,k+1}, m)$ be the new debt of the agent on the jth row in the $k+1$st column after applying payment pattern P to the kth column, where those agents initially start with debt $m = (m_1, m_2, \ldots, m_c)$. Let $\mathbb{P}(m)$ be the set of all valid column payment patterns for a column with debts $m = (m_1, m_2, \ldots, m_c)$. Finally, let $|P|$ be the number of transactions used in the payment pattern P. We have the following recursion

$$f(k, m_1, m_2, \ldots, m_c) = \min_{P \in \mathbb{P}(m)} \left(f\left(k+1, m_1', m_2', \ldots, m_c'\right) + |P| \right)$$

Final Answer: We calculate the final answer by calculating $f(1, w(v_{11}), w(v_{21}), \ldots, w(v_{c1}))$. This algorithm calculates the minimum number of transactions, but doesn't find the actual sequence of transactions. We can easily modify our DP to do that by keeping track of which $P \in \mathbb{P}(m)$ gave us the minimum.

Theorem 5.4.2. *Algorithm 6 correctly finds the optimal transactions to make on this fixed height graph*

Proof. Our goal here is to show that the optimal solution must use one of the column payment patterns that we iterate over. Just like the original debt settling problem, the problem of settling debt on a graph can be reduced to zero-sum set packing, where now we require that the sets we choose not only be zero-sum, but also connected in the graph. Figure 10 is a good example of this in action because we divided our grid into 4 disjoint connected components such that the sum of the debts in each component is 0. We can see that with those 12 nodes, it would normally take 11 transactions to settle the debt in the natural way, but we can do it in $12 - 4 = 8$ because there are 4 disjoint zero-sum components. It is clear that within each connected zero-sum component, it is possible to settle the debt by making each node the sender of a transaction at most once. Even if an amount sent is negative, we still use the call the first element of the transaction tuple the sender. Therefore, in the optimal solution, there must be a way for every node to be the sender of a transaction at most once. We exhaustively search for all such transaction patterns and thus we have are guaranteed to find an optimal solution through this DP. □

Theorem 5.4.3. *Algorithm 6 runs in $O(n(2M+1)^c 4^c)$ time.*

Proof. There are $\frac{n}{c}$ possible inputs for k, and $2M+1$ possible inputs for each of $m_1, m_2, \ldots, m_c$ because each can range from $-M$ to M. No matter how the debt is settled, no one will ever have debt or surplus whose absolute value is greater than M.

There are $O(4^c)$ possible column payment patterns for any given column. This is because each agent in the column can pay Up, Down, Right, or Nothing. A lot of these will not be valid for a debts $m_1, m_2, \ldots, m_c$ of the agents in the kth column.

Since this DP has $O(n(2M+1)^c)$ states and each state takes $O(4^c)$ time to calculate, we get that the overall running time is $O(n(2M+1)^c 4^c)$. This shows that algorithm 6 is a pseudopolynomial algorithm for the debt settling problem over a grids of constant height. □

We have shown already seen that the debt settling problem over a general graph is NP-complete because when G is a complete graph, we know it is NP-complete. We have just found a pseudopolynomial time algorithm for the debt settling problem when the underlying graph is a fixed width grid, so certainly this problem is not strongly NP-complete. But it is still an open problem whether settling debts on a fixed width graph is NP-Complete. Perhaps there exists a polynomial-time algorithm that we just haven't found yet.

5.5 Bounded Pathwidth

In this section, we will present a pseudopolynomial time algorithm for the debt settling problem when the underlying graph G has bounded pathwidth. We will show that in this case, we can use a similar dynamic programming algorithm as algorithm 6 to get a pseudopolynomial algorithm for finding the minimum set of transactions to make.

5.5.1 Definition of Pathwidth

Given a graph $G = (V, E)$, a path decomposition of the graph is a sequence $X_1, X_2, \ldots, X_t$ of subsets of V with the following properties:

- For each edge $e = (u, v) \in E$, there exists an X_i which contains both u and v.
- For any three indices $i \leq j \leq k$, we have $X_i \cap X_k \subset X_j$.

Example 5.5.1. Here is an example of a graph and a possible path decomposition:

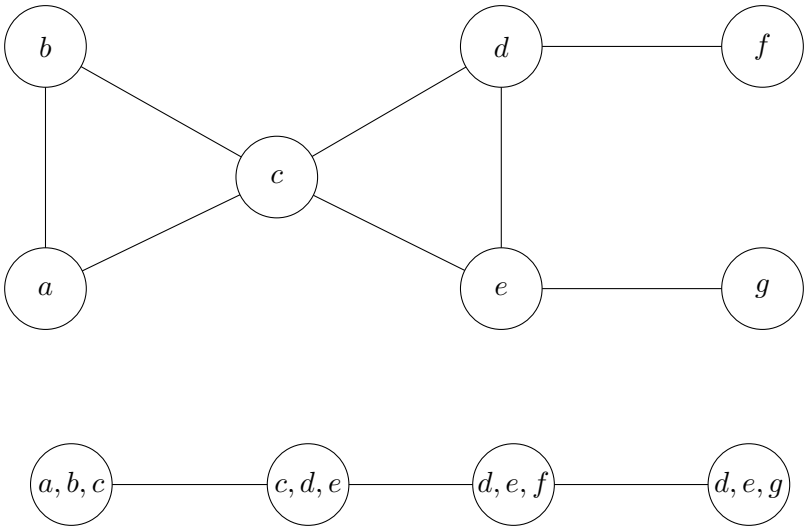


Figure 13: Example of Path Decomposition

The decomposition is $X_1 = \{a, b, c\}, X_2 = \{c, d, e\}, X_3 = \{d, e, f\}, X_4 = \{d, f, g\}$.

The width of a particular decomposition is $\max |X_i| - 1$. We subtract one so that the path width of an actual path is 1. The graph above has path width 2. Before we present the algorithm, we prove an important lemma that will help show the correctness of our dynamic programming algorithm and another lemma to help us analyze the running time of our algorithm:

Lemma 5.5.1. *If $v \in X_k$, but $v \notin X_{k+1}$, then $v \notin X_j$ for all $j > k + 1$ as well.*

Proof. This is easily proven via property 2 of the path decomposition. If $x_i \in X_k, x_i \notin X_{k+1}$ but $x_i \in X_{k+c}$ for $c > 1$, then property 2 would fail because $x_i \in (X_{k+c} \cap X_k)$ but if $x_i \notin X_{k+1}$, then how can $(X_{k+c} \cap X_k) \subset X_{k+1}$? Another way of stating this property is that if we look at $S_v = \{i : v \in X_i\}$, then S_v will be a contiguous subsubsequence of $\{1, 2, \ldots, t\}$. Once a node v is added to the X_i's and then removed, then it is will never appear in any subsequent X_i's again. □

Lemma 5.5.2. *For any graph G with pathwidth w, there exists a path decomposition of G where t, the number of X_is used in the decomposition, is $O(n)$.*

Proof. If we look at two adjacent sets of vertices X_i and X_{i+1} and we have that $X_i = X_{i+1}$, then we can remove X_{i+1} and the path decomposition properties are still satisfied. Therefore, moving from X_1 to X_t, at every step, we see that each X_{i+1} has at least 1 extra node or 1 less node than X_i. But we showed in 5.5.1 that once a node is added to the sequence X_i and then removed, it cannot reappear. In a fixed path decomposition, for each node v, there exists a unique index i where $v \notin X_{i-1}$ but $v \in X_i$ and another unique index $j \geq i$ where $v \in X_j$ but $v \notin X_{j+1}$. So, if our sequence $X_1, X_2, \ldots, X_t$ does not have any adjacent nodes being the same, then each index i. This means that between every two indices i and $i+1$, either a new node was added or a node was permanently deleted. Every node is added exactly once, and deleted exactly once, although multiple nodes can be added or deleted. But, an upper bound for t is certainly $2n$ and thus $t = O(n)$. □

5.5.2 Algorithm

Suppose we are given as input our graph $G = (V, E)$, a path decomposition $X_1, X_2, \ldots X_t$ where the pathwidth is p. Even if the path composition is not given, [6] has shown that it can be computed in polynomial time for a fixed p . Within each X_i, the number of ways in which the agents can pay each other is bounded by a function only depending on p. Our algorithm will start from X_1 in the path decomposition and perform all possible payments between agents in X_1. We will check that any node which is in $X_2 - X_1$ has their debt settled, and then recursively solve the problem where now the agents in X_2 may have different debts than before.

Algorithm 7 DP algorithm for fixed pathwidth graphs

Let $v_1, v_2, \ldots, v_n$ denote the nodes of the graph. Our recursive function $f(k, m)$ represent the best possible way to settle the debt for the nodes in $X_k \cup X_{k+1} \cup \ldots \cup X_t$ where the nodes in X_k start with debts given by m and nodes not in X_k start with their original debt given by $w(v)$. Here, we can think of m as a mapping from the nodes in X_k to integers representing the current debt of those nodes. For every other node which does not appear in X_k, we assume their debt is $w(v_i)$ as passed in the input.

Base Case: The base case is when $k = t$, the last node in the path. In this case, m tell us what the debts of the people in X_t are, and since $|X_t| \leq p + 1$, we can simply check all the ways for people in X_t to pay each other and see which ones settle the debt, which is a function of p. Some of these payment patterns may be thrown out because the nodes of X_t don't all have to be connected. Of the ones that are valid and settle the debt, we take the one with the smallest number of transactions. If it is not possible to settle the debt, then return that it will take an infinite number of transactions to settle the debt.

Recursive Case: For X_k, we will try **all** ways in which the agents of that node can pay each other, and all possible amounts from $-M$ to M to be paid for each transaction. Each set of transactions (including amount) on these agents in X_k is called a payment pattern. For each way payment pattern, we will get some new debts for the agents in X_k. Suppose $X_k = \{v_{i_1}, \ldots, v_{i_b}\}$ where $b \leq w + 1$. We are given as input $m = (m_{i_1}, m_{i_2}, \ldots, m_{i_b})$, which maps some of the elements of X_k to their debts. After executing a particular payment pattern, we will get $m' = (m'_{i_1}, m'_{i_2}, \ldots, m'_{i_b})$ which are the new debts of the agents in X_k. These debts are calculated by Now, if an agent v_i is in X_k, but not in X_{k+1}, then we must have that $m'_i = 0$ or else this this payment pattern is not valid. This is because by lemma 5.5.1, v_i cannot appear in any of the X_j's for $j > k$. Therefore, if v_i's debt is not settled now, then it will never be settled in future iterations.

Once we have a payment pattern among the X_k's that settles the debt for all v_i that appear in X_k but not X_{k+1}, we will get new debts for the agents in X_k. Let m' be a map from every vertex that is in $X_{k+1} \cap X_k$ to its new debt value after applying this payment pattern. We check all possible payment patterns among the agents in X_k and take the one that minimizes the sum of the number of transactions used in the payment pattern and number of transactions in $f(k+1, m')$.

Final Answer: To calculate the final answer, we need to evaluate $f(1, m)$ where m maps the vertices of X_1 to their original debts given by $w(v_i)$ for $v_i \in X_1$.

Theorem 5.5.3. *Algorithm 7 runs in pseudopolynomial time for a fixed p.*

Proof. There are t possible values for k, first input of f. We showed in lemma 5.5.2 that $t = O(n)$. The next input is m, which maps up to p vertices to new debt values. There are $2M + 1$ possible debt values, and so there are $(2M + 1)^p$ possible values of m for every fixed value of the first input k. Therefore, the number of states in this DP is $O(n(2M + 1)^{p+1})$.

In order to calculate each state, we try applying a particular payment pattern to the nodes in X_i. What is an upper bound on the number of payment patterns we can have? Each agent has the option of doing nothing or paying 1 of the p other agents. Therefore, an upper bound on the number of payment patterns there can be is $O(p^{p+1})$. Note that since we only have to settle the debt for those agents that do not appear in X_{i+1}, for each payment pattern, we have up to $(2M + 1)^{p^{p+1}}$ ways to assign amounts to those transactions. Then, it takes some $O(n)$ work to apply the payment pattern and check to see if it is valid. Therefore, it takes up to $O(np^{p+1}(2M + 1)^{p^{p+1}})$ time to calculate each state of f.

Since our goal was to just come up with a pseudopolynomial time algorithm for a fixed pathwidth, we use fairly loose bounds in the analysis of our algorithm. The overall run-time is the product of the number of states in this DP and the amount of time it takes to calculate each state of f. For a fixed value of p, we get that the run-time is poly(n, M) and hence algorithm 7 is a pseudopolynomial algorithm for the debt settling problem on bounded pathwidth graphs. □

This section generalized what we saw with fixed width grids in the previous section. A grid with height c has pathwidth $2c$. We can see this by making every two adjacent columns of the grid into one of the X_i's in the path decomposition. Therefore, we have found that a larger class of graphs, namely ones with bounded pathwidth, also exhibit pseudopolynomial time algorithms for finding the minimum sequence of transactions. Whether there exists a polynomial time algorithm for it is still an open question.

5.5.3 Reduction to Minimum Edge Max Flow

In this section, we will show how the debt settling problem over a graph can be reduced to a max flow problem where you are trying to the minimum edge max flow problem.

Definition 5.5.1. In the **Minimum-Edge Max Flow** problem, we are given a graph $G = (V, E)$, a source $s \in V$ and a sink $t \in V$. Each edge also has a nonnegative capacity. Our goal is to find an $s - t$ flow that achieves the maximum flow value over all $s - t$ flows on G and uses the minimum number of edges possible over all flows that achieve this flow value.

The best way to show this reduction is through an example. Suppose we have the following graph G that we're trying to settle debt over:

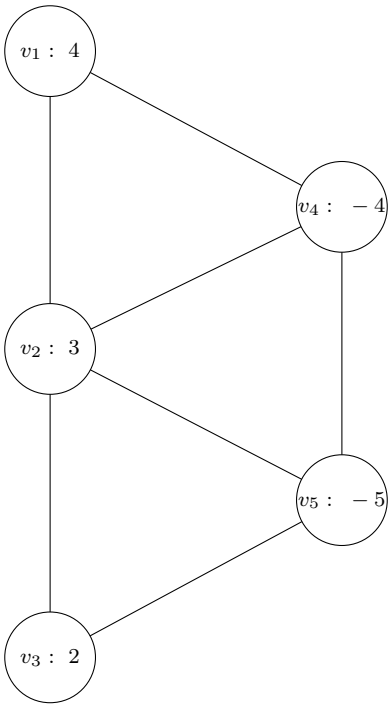


Figure 14: Input Graph for Minimum-Edge Max Flow Reduction Example

It isn't too hard to see that he minimum sequence of transactions to use is to have v_1 pay v_4 in 1 transaction and then have v_2, v_3, v_5 pay each other in some way to settle the debt in 2 more transaction. This can be reduced to a max-flow problem in the following way:

Algorithm 8 Reduce the debt settling problem over a graph to minimum edge max flow

We add a source s and a sink t to our graph G, and add the edge (s, v_i) with capacity $w(v_i)$ if $w(v_i) > 0$. Otherwise, if $w(v_i) < 0$, then we add the directed edge (v_i, t) with capacity $-w(v_i)$. Every other edge in G remains, but now with capacity ∞.

First, we can use any max flow algorithm to find the value of the $s-t$ maximum flow in this modified G. Note that because we assume G is connected, the value of the max flow will always be $\frac{1}{2}M$, where M is defined as before to be the sum of the absolute values of all the debts. Then, we are interested in the flow that achieves this maximum value that uses the fewest number of edges total.

The following is an example of how the graph from Figure ? can be converted into a max flow.

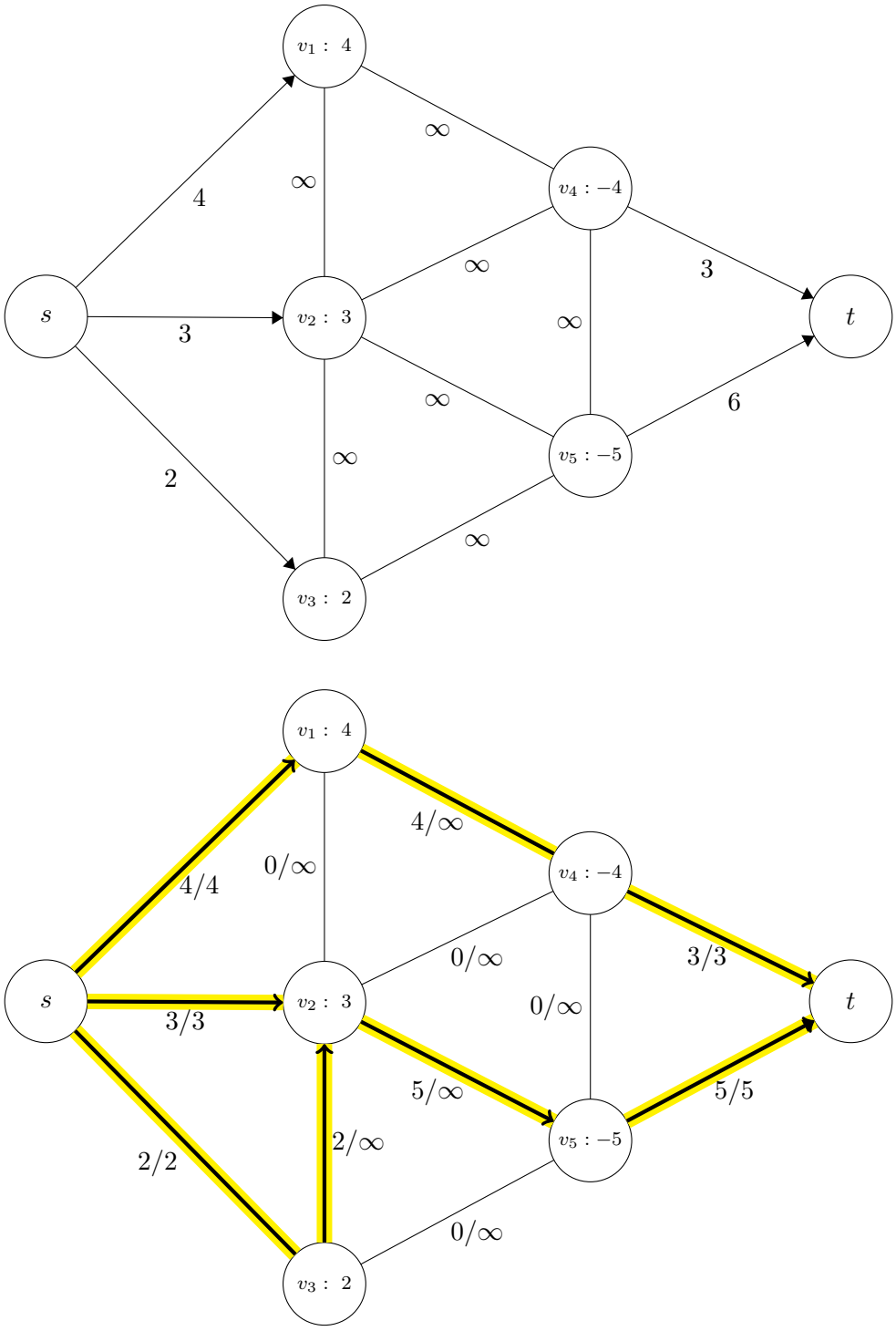


Figure 15: Example of Reduction to Minimum-Edge Max Flow

Theorem 5.5.4. *Algorithm 8 is a correct reduction of the debt settling problem over a graph to the minimum edge max flow problem.*

Proof. The main reason this reduction is correct is because the way we defined settling debt is very similar to conservation of flow. For example, a person with debt 10 has their debt settled if the

difference between the amount they pay other and the amount they receive from others is 10.

It is not difficult to formally verify that the properties of a flow imply that the debt settling equation (1.1) is satisfied, but we give some intuition as to why that is the case. In our max flow problem, if an agent has debt $w(v_i)$ where $w(v_i) > 0$, then we connect s and v_i with capacity $w(v_i)$. This means that in any flow solution which achieves the max flow of $\frac{M}{2}$, there will be $w(v_i)$ units of flow sent from s to v_i. By conservation of flow, we know that the number of units of flow leaving v_i must be $w(v_i)$ higher than the number of units of flow entering v_i besides those $w(v_i)$ units from s. v_i is not directly connected to t, so any other units of flow coming in or out must be to other agents. Therefore, if we treat units of flow as cash flow, a valid max flow will settle v_i's debt. A similar argument is true when $w(v_i) < 0$ and it is connected to t. □

Remark: The general debt settling problem over a complete graph can also be reduced to a minimum edge maximum flow problem in which every node that is not s or t is connected to each to each other.

Corollary 5.5.4.1. *The minimum edge max flow problem is NP-complete because we have just shown that the general debt settling problem reduces to it.*

This reduction also implies that it is possible to take an instance of the set packing problem and convert it into a minimum edge max flow problem:

Figure 16: Reducing Set Packing to Minimum Edge Max Flow

This is a surprising result because it shows that if we can solve the minimum edge max flow problem, then we can not only solve the debt settling problem, but also the general set packing problem. These two seemingly unrelated problems are connected through the debt settling problem.

Chapter 6

Conclusion

6.1 Overview of Thesis

We began this thesis by motivating and defining the **debt settling problem**, showed that there is a very natural solution to this problem and proposed a more efficient solution. We noticed that the number of zero-sum sets we could partition the debts into was inversely proportional to the number of transactions needed to settle the debt. This led us to define the **zero-sum set packing** problem. Both of these problems are shown to be NP-complete.

We continued by exploring approximation algorithms for the zero-sum set packing problem and the debt settling problem. For the zero-sum set packing problem, we proposed an algorithm which instead of taking exponential time to find all zero-sum sets, only found those up to a certain size constant k. This approximation algorithm does poorly in the worst case, but on experimental data generated from a probabilistic model, it performs quite well when we choose $k = 5$ and n is large enough. We also proposed an approximation algorithm for the minimum number of transactions. This algorithm is based on the $k-$set packing problem. We proposed a simple version of the algorithm, showed that it does not generalize for larger values of k, and then modified it to be able to generalize for any α_k approximation to the k-set packing problem.

Moving from algorithms to complexity, we asked the question of whether the zero-sum set packing problem is still NP-complete if we take out the bottleneck of having to find the zero-sum sets. This defined the $k-$zero-sum set packing problem, where we are not only given the debts, but also all the primitive zero-sum subsets of the debts of size at most k. We showed that the k-zero-sum set packing problem is NP-complete for $k \geq 3$ via reductions from maximal independent set and numerical 3-D matching. We also showed that the $n-$zero-sum set packing problem is just as hard to approximate as the regular set packing problem.

Then, we moved on to a variation of the debt settling problem where we are given a social graph and payments are only allowed between vertices connected by an edge. We saw that there is still a natural solution taking $n - 1$ transactions as long as the social graph is connected. When the graph is complete, this reduces to the regular debt settling problem, which is NP-complete. When the graph is a tree, we can solve it in polynomial time. When the graph is a fixed height grid or has bounded pathwidth, we can find pseudopolynomial time algorithms for solving the debt most efficiently. Finally, we looked at the reduction of the debt settling problem over a graph to minimum edge maximum flow problem. This interesting reduction allowed us to string together a series of

reductions from set packing all the way to minimum edge max flow.

6.2 Further Work

We have seen throughout this thesis that the debt settling problem gives rise to many interesting results in algorithms and complexity. Here are some of the areas for possible further research:

In chapter 3, we showed a $\frac{13}{10}$ approximation algorithm for the minimum number of transactions needed to settle all the debt. by using a $1, \frac{5}{3}$ and $\frac{1}{2}$ approximation to the $2, 3$ and 4 set packing problems. Further work can be done to figure out a closed form for the debt settling problem approximation when we are given α_k approximations for the k-set packing problem.

In chapter 4, we showed that the n-zero-sum set packing problem is just as hard to approximate as the general set packing problem, but we said that the same argument cannot be used to show that the k-zero-sum set packing problem is just as hard to approximate as the k-set packing problem. Therefore, it certainly is possible to find better than $\frac{5}{3}$ approximations for say the 3-zero-sum set packing problem that exploits the zero-sum structure of the subsets.

In chapter 5, we showed that the debt settling problem over a tree can be solved in polynomial time, while the debt settling over a fixed height grid or bounded pathwidth graph can eb solved in pseudopolynomial time. An interesting question is to find more precise graph classes for which the debt setting problem is in solvable in polynomial time, exhibits pseudopolynomial algorithms, and is strongly NP-complete.

Bibliography

[1] R.M. Karp, "Reducibility among Combinatorial Problems" *Proceedings of the third annual ACM symposium on Theory of computing*, pp.151–158, 1971.

[2] M. M. Halldórsson, "Approximating Discrete Collections via Local Improvements" in *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, (Philadelphia, PA, USA), pp. 160–169, Society for Industrial and Applied Mathematics, 1995.

[3] V.V. Micali, S.;Vazirani, "A $|V| * \sqrt{|E|}$ algorithm for finding maximum matching in general graphs", *"Symposium on Foundations of Computer Science"*, pp. 20–39, 1980.

[4] E. Hazan, S. Safra, and O. Schwartz, "On the Complexity of Approximating k-Set Packing", *Computational Complexity*, vol. 15, no. 1, pp. 20–39, 2006.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[6] Hans L. Bodlaender, "A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth", Society for Industrial and Applied Mathematics , 1995