Technische Universität Berlin
Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Algorithmics and Computational Complexity (AKT)

Resource Sharing: Reducing Envy Through Social Networks

Masterarbeit

von Florian Sachse

zur Erlangung des Grades „Master of Science“ (M. Sc.)
im Studiengang Computer Science (Informatik)

Erstgutachter:	Prof. Dr. Rolf Niedermeier
Zweitgutachter:	Prof. Dr. Markus Brill
Betreuer:	Dr. R. Bredereck, A. Kaczmarczyk, Prof. Dr. R. Niedermeier

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

Berlin, April 16, 2019

_____

Zusammenfassung

In vielen sozialen und technischen Systemen ist die optimale Verteilung von Ressourcen eine wichtige Aufgabe. Dabei hängt die Optimierung in vielen Fällen von den Beziehungen zwischen den Agenten der Systeme ab. In der vorliegenden Arbeit wird das Konzept *graph-envy*, d.i. *Neid* entlang eines sozialen Netzes, so erweitert, dass Agenten nun entlang dieses Netzes Ressourcen teilen können, um so die Verteilung der Ressourcen zu verbessern.

Für dieses Modell werden Algorithmen beschrieben, mit deren Hilfe die utilitarian social welfare und egalitarian social welfare von Verteilungen optimiert werden können. Es wird formal das NP-harte Problem der Reduzierung der Anzahl neidischer Knoten durch das paarweise Teilen von Ressourcen dargelegt und die Komplexität bezüglich mehrerer Parameter untersucht.

Weiterhin wird ein Greedy-Algorithmus dargestellt, mit dessen Hilfe das Problem in Linearzeit auf Pfaden gelöst werden kann. Ein ebenfalls berschriebener Algorithmus basierend auf dynamischer Programmierung schließlich löst das Problem auf Bäumen in polynomieller Zeit.

Abstract

Finding optimal resource allocations is an important task in many social and technical systems. The notion of optimality often depends on underlying connections between agents. We extend the idea of *graph-envy*, that is, envy along a social network, by allowing agents to share in pairs along a social network to improve an allocation.

Using this model, we present algorithms for optimizing utilitarian and egalitarian social welfare of allocations. We introduce the NP-hard problem of reducing the number of envious nodes through pairwise sharing of resources and examine the hardness with respect to several parameters.

Furthermore, we present a greedy algorithm solving this problem in linear time on paths and a dynamic programming algorithm solving the problem in polynomial time on trees.

Contents

1 Introduction

The classic problem of resource allocation and its extensive studies have generated numerous formal models and techniques. From the straightforward question of how to distribute given resources fairly to involved agents to more complex models that consider complex relations between resources and/or agents various scenarious have been examined. In this thesis we extend the work of Bredereck et al. [BKN18] by allowing agents to engage in the pairwise sharing of resources. The choice of pairwise sharing will be properly motivated in the next section and is providing a rich theoretical setting, which we explore in the following chapters.

In the remainder of this chapter, we provide background and motivation for our model followed by the formal preliminaries. We finish the chapter with a pointer to related work.

The remaining parts of this thesis is divided into five chapters. In Chapter 2 we provide insights on how social welfare is affected by the introduction of pairwise sharing. Chapter 3 contains hardness results for the minimization of envy. We then provide algorithms for the special graph classes of *paths* and *trees* in Chapter 4 and Chapter 5 respectively. Finally, we provide a discussion of the results and an outlook on future work in Chapter 6.

1.1 Motivation

Allocating resources to agents is one of the important central tasks in many social and technical systems. Consequently, it has been the subject of extensive studies and has been examined in the context of various applications. One such application is *housing allocation*, that is, the assignment of tenants to houses, as it occurs for example in the case of on-campus housing for college students. Various models have been used to study this type of allocation problem, focusing on various aspects. Hylland and Zeckhauser proposed strategies for assigning candidates to positions so that the result is *pareto-optimal* [HZ79]. Shapley and Scarf examined how trading preallocated indivisible resources, such as houses, may result in an optimal allocation.

Abdulkadiroğlu and Sönmez remark that the special case of allocation of on-campus housing is even more involved [AS99]. They note that agents are not only new students in need of accommodation but also senior students trying to improve their current housing situation by trading according to some personal evaluation. Using a model accounting for both types of agents, they devise algorithms to compute pareto optimal allocations in this setting.

It is necessary to emphasize how important optimizing on-campus housing allocations

is. As noted by Astin, students living in college residence halls are not only more satisfied with their undergraduate experience but are also more likely to aspire to a graduate degree [Ast84]. Additionally, these students are also more socially engaged. On the other hand, dissatisfaction with the living situation, and especially with a roommate, often results in more stress for the student, as was found by Dusselier et al. [Dus+05]. This suggests it might be beneficial to take social structures into account when optimizing on-campus housing allocations.

We also want to refer to the work of Festinger, who provides convincing evidences that in social groups, such as students on a college campus, people tend to evaluate their own satisfaction with for example the distribution of resources by comparison with their peers [Fes54]. This suggests that students satisfaction can often be related to the allocation of on-campus housing.

In addition, we also want to emphasize the special role of two-person dormitories (as they are common in the united states) in the context of housing allocation. The classic formulation of the problem considers houses to be indivisible, unshareable resources. This is clearly not the case for such dormitaries, that are always shared between two students. Building on the insights of Abdulkadiroğlu and Sönmez, we also want to distiniguish between senior students already living in a dormitory and freshman students in need of a housing solution [AS99]. When optimizing the allocation of dormitories to students, it may often be necessary to account for seniors already living in a dormitory room and others that can then share a room with these seniors. This situation may then even be seen as the optimization of an existing allocation through sharing.

We propose now in this thesis a novel approach to optimizing allocations of indivisible items by sharing over a social network. Agents in this model can engage in pairwise sharing of resources (such as two-person dormitories) to optimize the overall allocation. Our model is closely following and extending the work of Bredereck et al. [BKN18]. They use the concept of *graph-envy* — envy along a social network — and analyze the complexity of finding envy-free allocations. In this work we examine if and how introducing pairwise sharing along the social network can improve an existing allocation with respect to three different metrics.

The first two metrics — called *utilitarian social welfare*) and *egalitarian social welfare* — evaluate the total utility of all agents and the minimum utility of all agents. Intuitively, optimizing the first is akin to optimizing the average utility even at the cost of big inequalities in the distribution. Optimizing the second is effectively improving the situation of the agents with the least utility.

The third and, as we show, most challenging metric is counting the number of envious nodes. For most of this thesis we concentrate on reducing this number by allowing agents to share through a social network.

1.2 Related Work

Allocating resources efficiently and, in some metric, fairly is a longstanding problem. Hugo Steinhaus proposed a first formal model and algorithm for the fair allocation of

divisible resources in 1948 [Ste48]. He introduced the metaphor of a cake for divisible resources. The accordingly named *cake-cutting* problem was and still is actively researched in for example computer science and political science. Recent surveys provide an overview of that research [Pro13; Pro15].

In contrast to divisible resources, allocating indivisible resources to agents has also been the subject of extensive research and different models have been developed. Surveys on this kind of allocation problem can provide an overview [Bou+16][Mar17].

The issue of improving initial allocations has been examined before. Shapley and Scarf applied the idea of applying centrally organized trading of resources to optimize an allocation [SS74]. Damamme et al. proposed a decentralized version relying on pairwise swaps to optimize allocations [Dam+15]. The work of Gourvès et al. extends on this idea by allowing swaps only between nodes connected by a social network [GLW17]. The difference between their work and ours is that items are shared and not swapped in our model. Sequences of swaps allow items to move over several nodes, whereas shared items will remain with their original owner.

1.3 Preliminaries

In this section, we present the formal framework used throughout the thesis. Most of the definitions presented here are either common mathematical concepts, such as graphs, or taken from previous work on the topic, most notably allocations and the concept of graph-envy.

1.3.1 Graphs & Allocations

We start with the definitions of the networks connecting agents in our setup. We model these networks as directed graphs. A graph consists of *nodes* connected via *edges*. Depending on the type of connecetions between nodes a graph can be either undirected or directed.

Definition 1.1. A *graph* $G = (V, E)$ consists of a set V and a set $E \subseteq \{\{i, j\} \mid i, j \in V, i \neq j\}$. We call the elements of V *nodes* and the elements of E edges of G.

Definition 1.2. A *directed graph* $G = (V, E)$ consists of a set V and a set

$$E \subseteq V \times V \setminus \{(i, i) \mid i \in V\}.$$

We call the elements of V *nodes* and the elements of E *arcs* of G. For an edge $(i, j) \in E$ the nodes i and j are called the *endpoints* of e. For an The graph $G' = (V, E')$, where $E' = \{\{i, j\} \mid (i, j) \in E\}$, is called the *underlying undirected graph* of G.

It is often helpful to visualize a graph, instead of providing the sets V and E explicitly. We use circles containing labels to represent nodes. We use arrows between these circles to represent arcs. Figure 1.1 depicts a graph consisting of the nodes

$$V = \{v_1, v_2, v_3, v_4\}$$

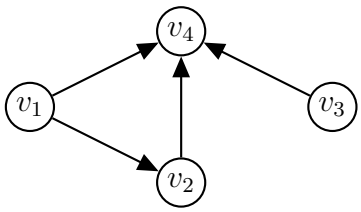


Figure 1.1: A simple example of a directed graph consisting of four nodes and four arcs.

and the arcs

$$E = \{(v_1, v_2), (v_1, v_4), (v_2, v_4), (v_3, v_4)\}.$$

Definition 1.3. Let $G = (V, E)$ be a directed graph and $i \in V$ a node. We then define the following sets.

- $N_{\text{in}}(i)$ is the set of neighbors of i connected to i via an *incoming* edge, that is,

$$N_{\text{in}}(i) := \{j \in V \mid (j, i) \in E\}.$$

- $N_{\text{out}}(i)$ is the set of neighbors of i connected to it via an *outgoing* edge, that is,

$$N_{\text{out}}(i) := \{j \in V \mid (i, j) \in E\}.$$

- $N(i)$ is the set of neighbors of i, that is,

$$N(i) := N_{\text{in}}(i) \cup N_{\text{out}}(i).$$

We say $i, j \in V$ are *adjacent* if $j \in N(i)$.

In the example in Figure 1.1, we have $N_{\text{in}}(v_2) = \{v_1\}$ and $N_{\text{out}}(v_2) = \{v_4\}$.

Next we provide a definition for allocation of *resources* to *agents*. While we do not enforce any structure on the set of agents yet (other than it being finite), we will later mostly consider allocations on the nodes of a given graph.

Definition 1.4. Let A, R be finite sets. An *allocation* of resources R to agents A is a function $\pi : A \to 2^R$ with

- for all $r \in R$ there is a $i \in A$ with $r \in \pi(i)$ and
- for all $i, j \in A$ it holds that $\pi(i) \cap \pi(j) = \emptyset$.

For $a \in A$ we call $\pi(a)$ the *bag* of a (under π).

The first condition ensures there are no unassigned resources, the second condition ensures no resource is assgined to more than one node.

$$\pi_{\mathbf{N}}(v_1) = \{r_1\}$$
$$\pi_{\mathbf{N}}(v_2) = \{r_2, r_3\}$$
$$\pi_{\mathbf{N}}(v_3) = \emptyset$$
$$\pi_{\mathbf{N}}(v_4) = \{r_4\}$$

$$\pi_{\mathbf{E}}((v_1, v_2)) = \{r_1, r_3\}$$
$$\pi_{\mathbf{E}}((v_1, v_4)) = \emptyset$$
$$\pi_{\mathbf{E}}((v_2, v_4)) = \{r_2\}$$
$$\pi_{\mathbf{E}}((v_3, v_4)) = \{r_4\}$$

Figure 1.2: An example sharing allocation for the graph in Figure 1.1.

1.3.2 Sharing Allocation

We now continue to define the new concept of *sharing allocations* of resources to the nodes and edges of a graph. Similar to an allocation, it assigns each node a set of resources. It also assigns some resources to arcs. Intuitively, we consider these resources shared by the two nodes of the arc.

Definition 1.5. Let $G = (V, E)$ be a directed graph and R a finite set of resources. A *sharing allocation* $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ of R on G consists of two functions $\pi_{\mathbf{N}} : V \to 2^R$ and $\pi_{\mathbf{E}} : E \to 2^R$ with

1. $\pi_{\mathbf{N}}$ being an allocation of resources R to agents V and
2. $\pi_{\mathbf{E}}((i, j)) \subseteq \pi_{\mathbf{N}}(i) \cup \pi_{\mathbf{N}}(j)$ for all $(i, j) \in E$.

We write

$$\pi(i) := \pi_{\mathbf{N}}(i) \cup \bigcup_{j \in N_{\text{in}}(i)} (\pi_{\mathbf{E}}((j, i)) \cap \pi_{\mathbf{N}}(j)) \cup \bigcup_{j \in N_{\text{out}}(i)} (\pi_{\mathbf{E}}((i, j)) \cap \pi_{\mathbf{N}}(j))$$

for $i \in V$.

Note that in any sharing allocation the nodes of the underlying graph are the agents for the allocation. We therefore use the terms interchangeably, when appropriate.

We provide in Figure 1.3 an example of a sharing allocation for the graph in Figure 1.1 to illustrate the previous definitions. As a first step we provide the allocation $\pi_{\mathbf{N}}$ of the resources $R = \{r_1, r_2, r_3, r_4\}$ to the nodes of the graph. All resources are assigned to a single node, so $\pi_{\mathbf{N}}$ is a valid allocation. To construct the sharing allocation $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ we define $\pi_{\mathbf{E}}$.

To check whether π is a sharing allocation, we need to make sure, that any resource assigned to an edge $(v_i, v_j) \in E$ by $\pi_{\mathbf{E}}$ is assigned to either v_i or v_j by $\pi_{\mathbf{N}}$. This is clearly the case for the definitions in Figure 1.3.

To understand, how π is describing the sharing of resources examine the nodes v_1 and v_2. We can see that $\pi_{\mathbf{E}}((v_1, v_2)) = \{r_1, r_3\}$ and thus now, that the two nodes are *sharing* the resources r_1 and r_3. We can be even more specific. Since $r_3 \in \pi_{\mathbf{N}}(v_2)$, we can say that v_2 is sharing r_3 with v_1 and, conversely, v_1 is sharing r_1 with v_2. The concept of original ownership (i.e. the notion of *which node* is sharing a resource) is important to determine if a sharing benefits a given node compared to the underlying allocation.

The following observation states that any allocation can be treated as a sharing allocation. This allows us to not differentiate between allocations and sharing allocations in later parts of the thesis.

Observation 1.6. *Any allocation π can be viewed as a sharing allocation $\mu = (\mu_{\boldsymbol{N}}, \mu_{\boldsymbol{E}})$ via*

$$\mu_{\boldsymbol{N}}(i) := \pi(i) \text{ for } i \in V \text{ and } \mu_{\boldsymbol{E}}(e) := \emptyset \text{ for } e \in E.$$

It will often be necessary throughout this document to talk about neighbors that a given node shares resources with. We therefore introduce some additional notation.

Definition 1.7. Let $G = (V, E)$ be a directed graph, R a finite set of resources and $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ a sharing allocation of R on G. We then define the following sets of nodes.

$$S_{\text{in}}(\pi, i) := \{j \in N_{\text{in}}(i) \mid \pi_{\mathbf{E}}((j, i)) \neq \emptyset\}$$
$$S_{\text{out}}(\pi, i) := \{j \in N_{\text{out}}(i) \mid \pi_{\mathbf{E}}((i, j)) \neq \emptyset\}$$

1.3.3 Simple 2-sharing Allocations

Sharing allocations as defined above are a rather expressive extension to the concept of allocations. In this thesis we mostly focus on a more restricted version that allows a node to share with no more than one other node and to only share a single resource. In later chapters we show that even such a restrictive form of sharing allows for rather interesting results.

We also argue that this restriction makes sense on a conceptual level. There are many types of resources that can be shared between exactly two parties. Consider for example two-person offices assigned as a reward or dorm rooms given to a portion of the freshman students. If seen as shareable resources, these examples can be expressed through *simple 2-sharing allocations* defined as follows.

Definition 1.8. Let $G = (V, E)$ be a directed graph, R a finite set of resources and $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ a sharing allocation of R on G. We call π a *simple 2-sharing allocation* of R on G if

1. $|\pi_{\mathbf{E}}(e)| \leq 1$ for all $e \in E$ and
2. $|S_{in}(\pi, i) \cup S_{out}(\pi, i)| \leq 1$ for all $i \in V$.

We call $\pi_{\mathbf{E}}((i,j))$ a *sharing* (between i and j) if the set $\pi_{\mathbf{E}}((i,j))$ is non-empty and introduce the following notation for these sharings. We write

- $i \overset{r}{\rightarrow}_\pi j$ if $(i,j) \in E/(j,i) \in E$, $r \in \pi(i)$ and $\pi_{\mathbf{E}}((i,j)) = \{r\}/\pi_{\mathbf{E}}((j,i))$,
- $i \rightarrow_\pi j$ if there is a $r \in R$ with $i \overset{r}{\rightarrow}_\pi j$,
- $i \not\rightarrow_\pi$ if there is no $j \in V$ with $i \rightarrow_\pi j$,
- $\not\rightarrow_\pi j$ if there is no $i \in V$ with $i \rightarrow_\pi j$, and
- $i \not\leftrightarrow_\pi$ if $\not\rightarrow_\pi i$ and $i \not\rightarrow_\pi$.

The first condition states that only one resource may be shared between any two nodes connected by an edge. The second condition restricts nodes to sharing with at most one neighbor.

We can see that the example sharing allocation π that we have introduced in Figure 1.3 is not a simple 2-sharing allocation since both conditions from Definition 1.8 are violated: v_1 and v_2 share two resources and v_4 is sharing with both v_2 and v_4. We give a proper example for a simple 2-sharing allocation after introducing *utilities* in the next subsection.

Simple 2-sharing allocations introduce an additional connection between a node and at most one of its neighbors. In that regard they introduce a *matching* on the underlying undirected graph. A *matching* on an undirected graph is a subset of edges that are pairwise disjoint. The connection between simple 2-sharing allocations and matchings is formalized in the following observation.

Observation 1.9. *The second condition in Definition 1.8 implies that exactly one of the following cases holds for a node $i \in V$.*

- *$j \rightarrow_\pi i$ for exactly one $j \in V$,*
- *$i \rightarrow_\pi j$ for exactly one $j \in V$, or*
- *$i \not\leftrightarrow_\pi$.*

Intuitively, Observation 1.9 states that a node is either sharing a resource, is being shared with or is not participating in any sharing. This means that any two sharings cannot have a node in common. Any simple 2-sharing allocation can therefore be seen as a matching consisting of all edges $e \in E$ for which $\pi_{\mathbf{E}}(e) \neq \emptyset$.

As stated in Observation 1.6 every allocation can be seen as a sharing allocation and even as a simple 2-sharing allocation. In later sections we often use allocations as a starting point to construct specific simple 2-sharing allocations. The following definition introduces the notion of extending a simple 2-sharing allocation.

Definition 1.10. Let $G = (V,E)$ be a directed graph, R a finite set of resources. Let $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ and $\pi' = (\pi_{\mathbf{N}}, \pi'_{\mathbf{E}})$ be simple 2-sharing allocations on G. For some $(i,j) \in E$ and $r \in \pi(i)$, we say π' extends π with $i \overset{r}{\rightarrow} j$ if

- $\pi'(k) = \pi(k)$ for all $k \in V \setminus \{j\}$ and
- $\pi_{\mathbf{E}}((i,j)) = \emptyset$ and $\pi'_{\mathbf{E}}((i,j)) = \{r\}$.

Definition 1.10 ensures that the result of an extension is a simple 2-sharing allocation. This means that we cannot extend with $i \xrightarrow{r} j$ if either i or j are involved in another sharing or if $r \notin \pi(i)$. As a result, we know that after extending π with $i \xrightarrow{r} j$ to get π' the resource r is now also part of j's bag:

$$\pi'(j) = \pi(j) \cup \{r\}.$$

Definition 1.8 describes our extended version of allocations as consisting of two resource assignments: one on nodes and another one on edges. The bag of resources of a single node then depends on not only the initially assigned resources but also the ones shared along the incident edges. Conversely, the bag of a node not shared with (i.e. there is no other node sharing with that particular node) will be the same as under the initial allocation. This property is formally stated in the following observation.

Observation 1.11. *Let $G = (V, E)$ a directed graph and $\pi = (\pi_{\boldsymbol{N}}, \pi_{\boldsymbol{E}})$ be a simple 2-sharing allocation on G. If for $i \in V$ we have $\not\rightarrow_\pi i$ then $\pi(i) = \pi_{\boldsymbol{N}}(i)$.*

Proof. Let $i \in V$ be a node that is not shared with, that is, $\not\rightarrow_\pi i$. We then have $S_{in}(\pi, i) = \emptyset$ and thus $\pi_{\mathbf{E}}((k,i)) = \emptyset$ for all $k \in N_{in}(i)$. From Definition 1.5 we get

$$\pi(i) = \pi_{\mathbf{N}}(i) \cup \underbrace{\bigcup_{k \in N_{\text{in}}} (\pi_{\mathbf{E}}((k,i)) \cap \pi(k))}_{=\emptyset} \cup \underbrace{\bigcup_{k \in N_{\text{out}}} (\pi_{\mathbf{E}}((i,k)) \cap \pi(k))}_{=\emptyset} = \pi_{\mathbf{N}}(i).$$

□

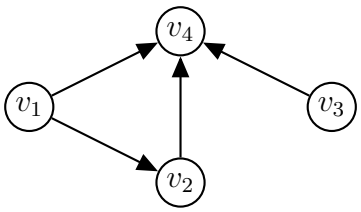
1.3.4 Utility

The definitions and observations up to this point fully define the underlying structure we want to examine in this work: resources being assigned to nodes and nodes sharing these resources with neighbors. Using this framework, we can describe simple 2-sharing allocations on different graph structures. As we strive to ultimately evaluate those allocations, we still need to provide a measure of quality. To this end, we introduce the concept of *utility*.

Definition 1.12. Let R be a finite set of resources. A function $u : R \to \mathbb{N}$ is called a *utility function*. For a set $R' \subseteq R$ we write $u(R')$ to denote the sum $\sum_{r \in R'} u(r)$.

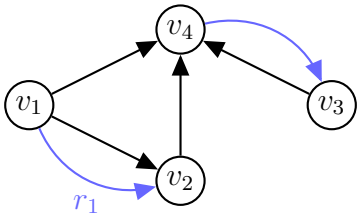
We extend the visualization of graphs to also illustrate simple 2-sharing allocations and utility functions. Figure 1.3 contains two examples of utility functions and (simple 2-sharing) allocations on the example graph presented above. We give the utility functions u_1,u_2,u_3 and u_4 of the nodes v_1,v_2,v_3 and v_4 respectively in a table next to the graph. An additional column indicates the node(s) that a resource is assigned to by the allocation.

	u_1	u_2	u_3	u_4	π^{-1}
r_1	1	0	0	0	v_1
r_2	1	1	1	2	v_1
r_3	1	1	1	2	v_3
r_4	1	1	1	1	v_4



(a) Utility functions and allocation π of resources r_1,r_2 and r_3.

	u_1	u_2	u_3	u_4	π^{-1}
r_1	1	0	0	0	v_1, v_2
r_2	1	1	1	2	v_1
r_3	1	1	1	2	v_3
r_4	1	1	1	1	v_4, v_3



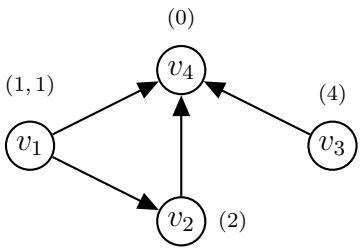
(b) A simple 2-sharing allocation extending π from the first example.

Figure 1.3: Visualization example for utilities and simple 2-sharing allocations.

Next to the table, we show the graph to be considered. As before, we use black arrows to indicate the directed edges. Additional blue arrows indicate sharings. They are annotated with the shared resource if there are multiple options. In the examples shown in the figure we provide an allocation in the first part and an extension containing sharings between v_1 and v_2 and between v_4 and v_3. Note that the sharings are also reflected in the table.

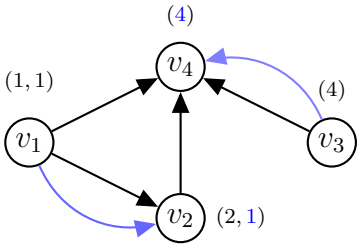
We often assume identical utility functions for all nodes of a graph. In this case we use a more compact visualization. Instead of providing the resource allocation in a separate table we annotate nodes with the utilities of the assigned resources. This representation emphasizes that we do not care about the names or labels of resources but are rather interested in the utility they provide.

In the following example, we again present the graph given in Figure 1.1 with an initial allocation. Under that allocation, the bag of v_1 contains two unit value resources. The bags of v_2 and v_3 each contain a resource of utility 2 and 4 respectively. We annotate the node v_4 with (0) to illustrate there are no resources assigned to it (and therefore no initial utility).



We also use blue arrows in this visualization to represent sharings. Additional utility through shared resources is added to the node annotations and written in blue. Note that

as we do not care about the name of the resources we do not annotate the blue arrows. The following example shows the above allocation extended with sharings between v_1 and v_2 and between v_3 and v_4.



The goal of this thesis is to improve initial allocations via sharing. This requires some metric of the quality of simple 2-sharing allocations. The following definition describe two types of metrics — generally called *social welfare* — that consider the allocation in its entirety. We do not judge the results of individual nodes but rather of the whole set of nodes.

Definition 1.13. Let $G = (V, E)$ be a directed graph, R a set of resources, $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ a simple 2-sharing allocation of resources in R on G and $\{u_i\}_{i \in V}$ a family of utility functions. The *utilitarian social welfare* $\mathrm{ut}(\{u_i\}_{i \in V}, \pi)$ of π is defined as

$$\mathrm{ut}(\{u_i\}_{i \in V}, \pi) := \mathrm{ut}(\pi) := \sum_{i \in V} (u_i(\pi(i))).$$

The *egalitarian social welfare* $\mathrm{eg}(\{u_i\}_{i \in V}, \pi)$ of π is defined as

$$\mathrm{eg}(\{u_i\}_{i \in V}, \pi) := \mathrm{eg}(\pi) := \min_{i \in V} (u_i(\pi(i))).$$

Utilitarian and egalitarian social welfare have been extensively studied in the context of resource allocation. We refer to Chevaleyre et al. for an overview of the usual definition and related results [Che+06]. Our definitions — while similar in shape to those in previous works — take into account the resources shared over edges.

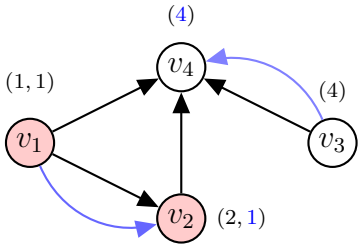
The second type of quality measure is related to a more local property of nodes: envy. In essence a node is envious if any of its (incoming) neighbors are assigned more valuable resources. This particular notion of envy is based on the works of Bredereck et al. Abebe et al. and Chevaleyre et al. among others [AKP17; BKN18; CEM17].

Definition 1.14. Let $G = (V, E)$ be a directed graph, R a finite set of resources and $\{u_i\}_{i \in V}$ a family of utility functions. Let $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ be a simple 2-sharing allocation. A node $i \in V$ is called *envious* (in π) if there is a $j \in N_{out}(i)$ with $u_i(\pi(i)) < u_i(\pi(j))$. We define the set of envious nodes in π as

$$\mathrm{Env}(\pi) := \{i \in V \mid \exists j \in N_{out}(i) : u_i(\pi(i)) < u_i(\pi(j))\}.$$

The concept of envy can now be used to determine the quality of a given simple 2-sharing allocation. In this work, we explore the possibility to influence the total number of envious nodes in a given allocation by introducing sharing. In particular, we aim to minimize the total number $|\operatorname{Env}(\pi)|$ of envious nodes.

We extend the visualization presented so far to show envious nodes marked in red. This would result in the following depiction of envious nodes in the previous example (as both v_1 and v_2 are envious of v_4).



2 Social Welfare

As discussed in Chapter 1, we want to determine if and how pairwise sharing can be used to improve existing allocations. We already introduced the formal framework to extend allocations. However, we still need to formalize when extensions actually constitute an improvement. Starting with this chapter, we therefore look at several metrics that are commonly used to compare quality (or fairness) of allocations.

In the following sections, we look into *social welfare* concepts. In particular, we study the possibility of using sharing to improve allocations with respect to the *utilitarian social welfare*, that is, the accumulated utility of all agents, and the *egalitarian social welfare*, that is, the minimum utility of any agent. Both metrics have been commonly used to evaluate the quality of resource allocations and are therefore obvious candidates for our attempt to introduce sharing to existing distributions.

Throughout this chapter, we consider a fixed directed graph $G = (V, E)$ and a family of utility functions $\{u_i\}_{i \in V}$ for a fixed set of resources R.

2.1 Utilitarian Social Welfare

Definition 1.12 shows how we assign numerical utility values to resources. Definition 1.4 describes how allocations assign sets of resources to nodes. When put together, these two definitions give us a clear understanding of how to determine a (numerical) utility value for any single node. While there is definitely more than one way to combine these into a utility value for the whole allocation, a rather straightforward one is to simply compute the sum of the individual node-utilities.

This metric for judging the quality of allocations, which is called the utilitarian social welfare and defined in Definition 1.13, is in fact well-established and has been studied in various contexts before (see Chevaleyre et al. for an overview [Che+06]). In this section we present some insights we then use to devise a simple algorithm to compute a simple 2-sharing allocation from an initial allocation that maximizes utilitarian social welfare.

2.1.1 On Optimality of the Simple 2-Sharing Allocations

We have already discussed in the previous chapter that we aim to extend existing allocations through the introduction of pairwise sharing. This task is strictly different from finding an optimal simple 2-sharing allocation (with respect to utilitarian social welfare). In fact, the best solution that can be computed from a given initial allocation might be significantly worse than other allocations.

	u_1	u_2	u_3	u_4	π^{-1}
r_1	2	1	2	4	v_2
r_2	4	2	1	2	v_3

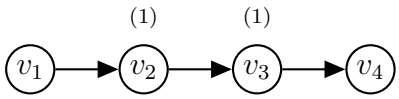


Figure 2.1: Non-optimal resource allocation for utilitarian (and egalitarian) social welfare.

Consider the example given in Figure 2.1 for illustration. It shows a graph consisting of 4 nodes, with each node having a custom utility function to evaluate the two available resources. The initial allocation assigns these resources to v_2 and v_3.

In this example the initial allocation assigns the resources r_1 and r_2 to v_2 and v_3 respectively. For these nodes however, the utility of the assigned resources is rather low. Even sharing these resources with one of their neighbors will generally improve the total utility by a small amount. In fact, it is easy to see that just by sharing we cannot improve the total utility by more than

$$u_1(r_1) + u_2(r_1) + u_3(r_2) + u_4(r_2) = 2 + 1 + 1 + 2 = 6.$$

In contrast, we can envision a different initial allocation π that assigns r_2 to v_1 and r_1 to v_4. This allocation would already have a much higher total utility:

$$u_1(r_2) + u_4(r_1) = 4 + 4 = 8.$$

Sharing the resources could improve this value even further, making this solution better than every sharing extending the given initial allocation.

2.1.2 Optimizing Utilitarian Social Welfare via Pairwise Sharing

We now show how to optimize utilitarian social welfare by extending an initial allocation via pairwise sharing. This is rather easy to do by reducing the problem to finding an optimal matching.

The key insight is that when extending a given simple 2-sharing allocation $\pi = (\pi_{\mathbf{N}}, \pi_{\mathbf{E}})$ with a sharing $v \xrightarrow{r} w$ to construct a new simple 2-sharing allocation $\pi' = (\pi_{\mathbf{N}}, \pi'_{\mathbf{E}})$, the change in utility is independent of π:

$$\begin{aligned}
\mathrm{ut}(\pi') - \mathrm{ut}(\pi) &= \sum_{v' \in V} \pi'(v') - \sum_{v' \in V} \pi(v') \\
&= \sum_{v' \in V} \pi'(v') - \pi(v') \\
&= \pi'(w) - \pi(w) \\
&= u_w(r)
\end{aligned}$$

This means that the order in which we consider sharings does not matter. When starting from an initial allocation π_0, we then can compute the utilitarian social welfare

of any simple 2-sharing allocation $\pi = (\pi_0, \pi_{\mathbf{E}})$ extending π_0 as

$$\mathrm{ut}(\pi) = \mathrm{ut}(\pi_0) + \sum_{\substack{(v,w)\in E \\ r\in\pi_{\mathbf{E}}((v,w))\cap\pi(v)}} u_w(r) + \sum_{\substack{(w,v)\in E \\ r\in\pi_{\mathbf{E}}((w,v))\cap\pi(v)}} u_w(r).$$

Optimizing utilitarian social welfare is thus equivalent to optimizing the sum

$$\sum_{\substack{(v,w)\in E \\ r\in\pi_{\mathbf{E}}((v,w))\cap\pi(v)}} u_w(r) + \sum_{\substack{(w,v)\in E \\ r\in\pi_{\mathbf{E}}((w,v))\cap\pi(v)}} u_w(r).$$

This shows that given an edge $(v, w) \in E$ in an optimal simple 2-sharing allocation π either no resource is shared between v and w or the optimal resource to share is the r maximizing

$$\max\left(\max_{r\in\pi_{\mathbf{N}}(v)} u_w(r), \max_{r\in\pi_{\mathbf{N}}(w)} u_v(r)\right).$$

Note that we now know the contribution of each edge that is shared over to the utilitarian social welfare of an optimal simple 2-sharing allocation. Finding the optimal simple 2-sharing allocation is thus equivalent to finding the optimal set of edges to share over. Observation 1.9 illustrates that this in turn is identical to finding a maximum weight matching. This gives us the main result of this section in the following theorem.

Theorem 2.1. *Finding a simple 2-sharing allocation extending a given allocation that maximizes the utilitarian social welfare can be done in*

$$\mathcal{O}\left(|R||E| + |V|\left(|E| + |V| \cdot \log(|V|)\right)\right)$$

steps.

Proof. As suggested we can reuse results for maximum weight matching (MWM). MWM is a problem from graph theory and has been extensively studied in the past. Edmonds published an algorithm, able to solve the unweighted version of the matching problem in $\mathcal{O}(|V|^2|E|)$ time, in 1965 [Edm65]. A slightly modified version that can solve the weighted version in the same asymptotic running time, has for example been implemented by Kolmogorov [Kol09]. Gabow improved this even further and provided an algorithm that solves the weighted matching problem on general graphs in $\mathcal{O}(|V|(|E| + |V|\log(|V|))$ [Gab18].

To facilitate the algorithm, we need to compute the edge weights. From the explanation above we know the weight for an edge $(v, w) \in E$ can be computed (in $\mathcal{O}(|R|)$ time) as

$$\max\left(\max_{r\in\pi_{\mathbf{N}}(v)} u_w(r), \max_{r\in\pi_{\mathbf{N}}(w)} u_v(r)\right).$$

□

2.2 Egalitarian Social Welfare

In the previous section we have described how summing the utility nodes assign to their respective bags can can be used to construct a metric for allocations. There are some obvious drawbacks to that approach, such as the fact that assigning all resources to a single node could still yield an optimal solution. It is thus interesting and useful to look into other metrics as well.

Another metric that combines the utilities of individual nodes is called *egalitarian social welfare*. Similar to the utilitarian social welfare, it has also been studied extensively (again we refer to Chevaleyre et al. for an overview [Che+06]). Instead of the sum, it uses the minimum node utility as the utility of an allocation. See Definition 1.13 for the definition of the egalitatian welfare $\mathrm{eg}(\pi)$ of a simple 2-sharing allocation π.

In this section we devise an polynomial time algorithm, computing a simple 2-sharing allocation that extends a given initial allocation optimally with respect to the egalitarian social welfare.

2.2.1 On Optimality of the Simple 2-Sharing Allocation

We again note that the task of extending an initial allocation via pairwise sharing to optimize the egalitarian social welfare is strictly different from finding an optimal simple 2-sharing allocation. And, as for utilitarian social welfare, the solution to our problem is often worse than the solution to the general optimization problem.

The setup presented in Figure 2.1 can also be used as an illustrating example for the egalitarian social welfare. With the two resources originally assigned to v_2 and v_3 only one of them can receive the other resource via sharing to increase its utility. Also, only by sharing with v_1 and v_4 respectively, v_1 and v_4 could get a non-zero utility. Thus, we can never increase the minimum utility above

$$\min(u(v_1), u(v_2), u(v_3), u(v_4)) = \min(2, 1, 1, 2) = 1.$$

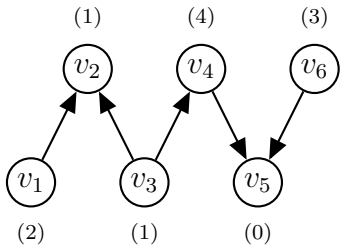
Observe that having initially r_2 assigned to v_1 and r_1 assigned to v_4 we could extend the allocation with $v_1 \xrightarrow{r_2} v_2$ and $v_4 \xrightarrow{r_1} v_3$. This would increase the minimum utility to

$$\min(u(v_1), u(v_2), u(v_3), u(v_4)) = \min(4, 2, 2, 4) = 2.$$

2.2.2 An Algorithm for Optimal Egalitarian Social Welfare

In this subsection we present an algorithm computing the optimal egalitarian social welfare one can achieve by extending a given allocation.

Note that while we still need to find the appropriate edges to share over, it is not obvious how we could translate this to finding a matching, like we did for utilitarian social welfare. Consider the following example.



Of the 5 nodes v_5 has initially the lowest utility. To optimize the egalitarian social welfare we would need to increase its utility. Both v_4 and v_6 could share their respective resource with v_5. It might be tempting to always go for the resource with the greater utility. In this example, however, this would not be optimal. By having v_4 share its resource with v_5, only v_2 could share with v_3. The resulting minimum utility would be

$$\min(u(\pi(v_1)), u(\pi(v_2)), u(\pi(v_3)), u(\pi(v_4)), u(\pi(v_5)), u(\pi(v_6))) = \min(2, 1, 2, 4, 4, 3) = 1.$$

If instead v_6 shared with v_5, then v_4 would now be free to share with v_3 and v_1 could then share with v_2. The minimum utility of any node then is

$$\min(u(\pi(v_1)), u(\pi(v_2)), u(\pi(v_3)), u(\pi(v_4)), u(\pi(v_5)), u(\pi(v_6))) = \min(2, 3, 5, 4, 3, 3) = 2.$$

This example illustrates that we cannot easily adapt the approach from the previous section to work for egalitarian social welfare. We instead present in Figure 2.2 an algorithm to compute the best possible value when extending a given allocation. We split the functionality in two functions: CanImproveMinUtility and MaxMinUtility. Intuitively, the former one can be used to check if it is possible to introduce sharings that improve the utility of all nodes above a given threshold. The latter function then uses binary search to determine the greatest threshold, for which the former still returns true.

To determine if we can improve the utility of all nodes above a given threshold d in an initial allocation π, CanImproveMinUtility(π,d) first determines the nodes that need to be updated. In line 5 it sets L to be the set of all nodes with utility below the threshold. It then proceeds to construct a set of edges M that one can share over to improve the utility of the nodes in L. The loop in line 7 iterates over all edges connecting nodes in L with those not in L. In line 8 the algorithm checks whether the current edge e can be used to increase the utility of the endpoint in L above the threshold. If this is true, the edge is then added to M. After all edges have been processed the algorithm proceeds in line 12 to check if the graph (V, M), that is, the graph containing only the collected edges, contains a matching covering all nodes in L. It returns true if this is the case and false otherwise.

With this algorithm we get the following theorem as our main result in this section.

Theorem 2.2. *Let π be an allocation and $b := \max_{i \in V} u_i(\pi(i))$. Then* MaxMinUtility*($\pi$) computes the maximum egalitarian welfare of simple 2-sharing allocations extending π in $\mathcal{O}\left(\log(b) \cdot (|R| + \sqrt{|V|}) \cdot |E|\right)$ time.*

Proof. Let π be an allocation of resources R to nodes V.

```
1: function CanImproveMinUtility(π, d)
2:     if min_{i∈V} u_i(π(i)) ≥ d then
3:         return True
4:     end if
5:     L ← {i ∈ V | u_i(π(i)) < d}
6:     M ← ∅                                   ▷ (V, M) is always a bipartite graph
7:     for all i ∈ L, j ∈ V \ L, e ∈ E ∩ {(i, j), (j, i)} do
8:         if max_{r∈π(j)} u_i(r) + u_i(π(i)) ≥ d then
9:             M ← M ∪ {e}
10:        end if
11:    end for
12:    if there is a matching in (V, M) covering L then
13:        return True
14:    else
15:        return False
16:    end if
17: end function

18: function MaxMinUtility(π)
19:    m ← 0
20:    M ← max_{i∈V} u_i(π(i))
21:    while m < M do
22:        if CanImproveMinUtility(π, ⌈(M−m)/2⌉) then
23:            m ← ⌈(M−m)/2⌉
24:        else
25:            M ← ⌈(M−m)/2⌉ − 1
26:        end if
27:    end while
28:    return M
29: end function
```

Figure 2.2: Listing of the MaxMinUtility algorithm to compute the optimum egalitarian welfare.

Correctness Let $d \in \mathbb{N}$. In a call of CanImproveMinUtility(π,d) L contains all nodes with utility lower than d (line 5). The loop then iterates over all edges connecting a node in L with one not in L (line 6) and updates M. We then know the following two properties of M, once the loop has finished (line 11).

- M only contains edges connecting a node in L with a node not in L. This is ensured by the condition in line 7. From this we get that (V, M) is bipartite with partitions L and $V \setminus L$.
- Each edge $(i, j) \in M$ supports the sharing of a resource with high enough utility to increase the utility of the node in L to at least d.

Let now $M' \subseteq M$ be a matching in (V, M). We construct π' by extending π with $i \xrightarrow{r} j$ for each $i \in L, j \in V \setminus L$ with $(i, j) \in M'$ or $(j, i) \in M'$ where $r = \arg\max_{r \in \pi(i)} u(i)$. The second property then ensures $u(\pi'(j)) \geq d$. If the matching covers all nodes in L, then this implies $\mathrm{eg}(\pi') \geq d$ and we can construct a simple 2-sharing allocation from the selected edges. Note that Observation 1.9 implies such a matching always exists if there is a corresponding simple 2-sharing allocation.

Finally, the binary search in MaxMinUtility ensures we return the greatest d, such that there is a simple 2-sharing allocation extending π for which the egalitarian welfare is at least d.

Running Time The loop in line 7 runs at most $\mathcal{O}(|E|)$ times. Within each iteration all resources of a particular node are checked, so the loop runs in $\mathcal{O}(|R||E|)$ time. Micali and Vazirani demonstrated how a maximum matching in a bipartite graph can be found in $\mathcal{O}(\sqrt{|V|}|E|)$ time [MV80]. Put together this implies a running time of $\mathcal{O}(|V| + |R||E| + \sqrt{|V|}|E|)$ for CanImproveMinUtility.

Finally, the binary search in MaxMinUtility requires $\lceil \log(\max_{i \in V} u_i(\pi(i)) \rceil$ steps. □

3 Hardness of Minimizing Envy Through Pairwise Sharing

In the previous chapter we have looked at two important and well-understood metrics for allocations built around the concept of social welfare. These metrics have been designed to derive a global utility value of an allocation from the utilities for every agent. As this effectively reduces the information at hand to just a single number, it is not hard to see, why this might not be an appropriate representation for every application.

In some resource allocation scenarios, such as reward assignment for employees or on-campus housing allocation for students, there are social structures to consider. We already argued in the introduction that student stress-levels seem to be related to their housing situation and the relation to their roommate (as for example shown by Dusselier et al. [Dus+05]). And as for example Festinger argues it is the case for most social groups, students and employees evaluate their own situation in part by comparison with others [Fes54]. It thus seems fitting to account for this fact when optimizing allocations.

The concept of *envy* has been introduced and studied in the context of allocations to model the comparison of agents with others. We again refer to Chevaleyre et al. for an overview [Che+06]. The idea is to mark agents as envious if they observe that they have been assigned a bag of lower utility than the ones assigned to other agents. Acknowledging that there are scenarios where agents only compare their own situation to a subset of other agents, such as the colleagues in their own team, Bredereck et al. Abebe et al., Chevaleyre et al. and others extended the concept of *envy* to adhere to an underlying social network [AKP17; BKN18; CEM17]. Definition 1.14 presents this notion of *graph-envy*, simply called *envy* throughout this thesis for simplicity.

In this chapter we investigate the complexity of minimizing envy when introducing sharing in allocations. One of the main results of this thesis is a hardness property for the resulting problem.

3.1 Graph Envy

While the classic problem of resource allocation is only concerned with agents their preference over resources, often some underlying structure can be observed in real-world applications. In technical systems not all agents might be able to interact. And in social settings there are often social networks influencing the actual impact of any distribution of resources.

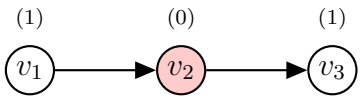
This aspect of interaction can often be captured by graphs. This is the foundation of works by Bredereck, Kaczmarczyk and Niedermeier [BKN18]. They used the concept

of graph envy. A graph relating the agents restricts what other agents they might be envious of. Building upon this general setup, we now ask if and how the number of envious nodes can be reduced in an allocation, when they are extended by allowing nodes to share in pairs.

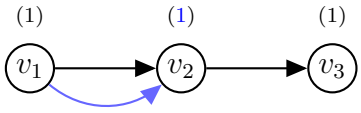
3.1.1 Spreading Envy

As described in Definition 1.5 our particular version of allocations prevents nodes from sharing with more than one neighbor and, hence, induces a matching on the considered graph (see Observation 1.9). Thus, it stands to reason that the process of finding a good simple 2-sharing allocation is similar to finding a good matching. We show below that this is not the case when optimizing envy and that this particular task is indeed harder than finding a matching.

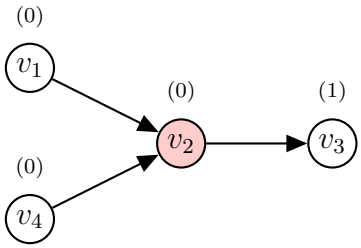
In the context of graph-envy an agent or node is envious if it can observe one of its neighbors having more utility accumulated with the assigned resources than he has himself, according to his own evaluation. The nodes he *can observe* is here given by a directed graph. We might now rightfully expect envious nodes to be contempt with their situation if their total utility could be improved by sharing resources with one of their neighbors. Consider for example the following small graph and allocation.



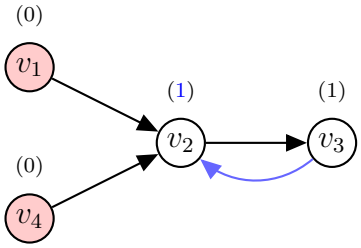
Under this initial allocation v_2, with no resources assigned to it, is envious of v_3, which has a 1-value resource allocated to it. However, by making v_1 share its 1-value resource with v_2, there is no longer any need to be envious.



By sharing the single resource of v_1 with v_2 we could construct an envy-free allocation and arguably improve the situation for some nodes without worsening it for others. This simple example already illustrates why it is interesting to look at opportunities to fix envy in a given allocation and graph by sharing resources. It is, however, also deceiving in its simplicity. Consider the following slightly modified version of the above example.

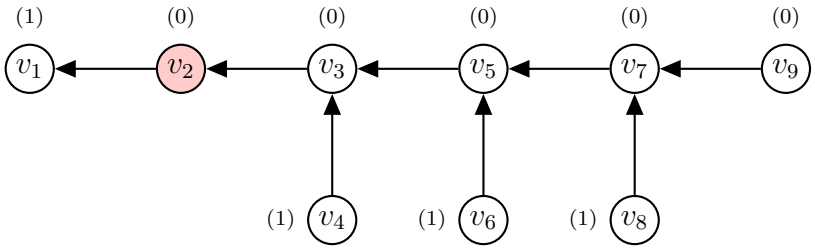


Here v_2 is starting off envious of v_3 as well. While v_1 now has no resource to share, it is still possible to fix the envy by making v_3 share with v_2.

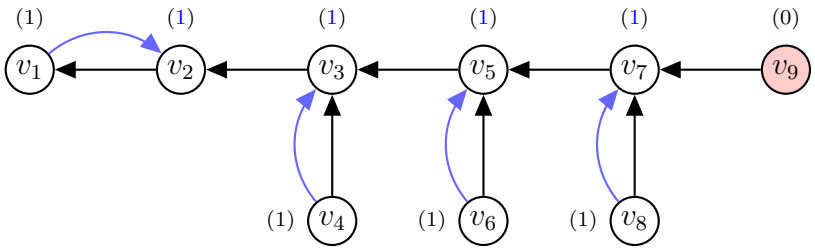


Doing this does indeed ensure that v_2 is no longer envious. However, as this is achieved by increasing v_2's utility, now both v_1 and v_4 are envious of v_2. Even worse, as v_2 is their only neighbor and since we do not allow a node to be involved in more than one sharing, this leaves both v_1 and v_3 envious.

The previous example shows an important property of envy, when we extend existing allocations via sharing: it can be spread through the graph. This is a rather striking difference compared to the optimization of social welfare, because locally optimizing the number of envious nodes can actually worsen the global situation. When we want to reduce the number of envious nodes in a given allocation by sharing, we need to be aware of the possibility of introducing envy in previously unenvious nodes. We might of course hope that these new envious nodes can then be made unenvious in a second step. However, it is not hard to construct an example that demonstrates how the spreading can propagate even further than just to the direct neighbors of a sharing node. Consider the following extended example with nine nodes.



Here v_2, being envious of v_1, is the only envious node. Again, we can easily fix the envy by making v_1 share with v_2. However, as with the previous example this now makes v_3, the other neighbor of v_2, envious. In contrast to the previous example, it is now possible to fix v_3 by having v_4 share with it. This sharing in turn makes v_5 envious. Continuing this chain reaction, we might end up with the following sharing.



Even though, we introduced all these sharings we ultimately were not able to reduce the number of envious nodes in the allocation.

It is important to realize that is not possible to determine whether the sequence of sharings ultimately leaves envious nodes by just looking at a constant number of nodes. The example above could be easily expanded to either leave no or any number of envious nodes in the final configuration.

Our main goal is to devise fast algorithms to compute sharings that decrease the number of envious nodes. While we cannot (yet) rule out the possibility of fast algorithms existing, the last example illustrates that the dynamics introduced by sharings moving envy in the graph complicates the endeavor quite a bit.

3.1.2 NP-hardness

As demonstrated in the previous section the task of extending an allocation via sharing has interesting dynamics regarding the number of envious nodes. In particular the examples show that envy can be moved or spread. Thus, when aiming to construct an optimal simple 2-sharing allocation, it is not obvious for single arcs if it is beneficial to share along them or not.

We are now interested in what bounds exists on the hardness of the problem. Formally, we introduce the following decision problem.

Graph Envy With Pairwise Sharing
Input: A Directed graph G, a finite set of resources R, an allocation π of R on G, $k \in \mathbb{N}$.
Question: Is there a simple 2-sharing allocation $\pi' = (\pi, \pi'_{\mathbf{E}})$ extending π with $|\operatorname{Env}(\pi')| \leq k$?

This problem and its corresponding optimization problem are motivated by our sharing setup and the examples presented so far. However, as we discuss in detail in section Section 3.2 we can focus on even more restricted versions of the problem for interesting results.

The following theorem — the main result of this section — formally states that there is indeed no algorithm that efficiently finds simple 2-sharing allocations with minimum envy (unless $P = NP$).

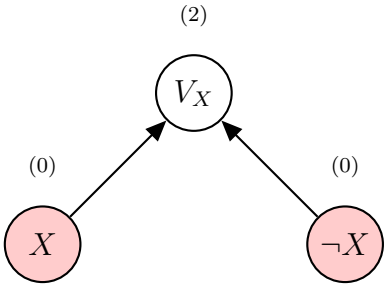
Theorem 3.1. Graph Envy With Pairwise Sharing *is NP-hard, even when we fix k.*

Proof. We provide a reduction from the well-known NP-hard problem 3-Sat. For an introduction to the problem and a proof of its hardness we refer to Kleinberg and Tardos [KT06, pp. 459].

3-Sat
Input: A boolean formula ϕ in 3-CNF form.
Question: Is there a truth-assignment such that ϕ is true?

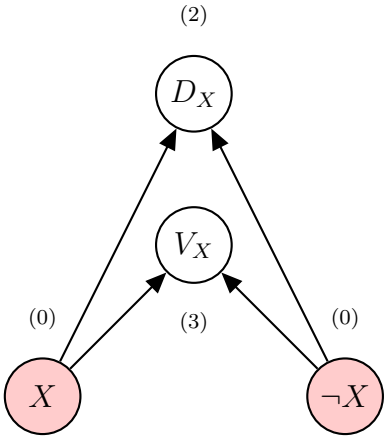
Let ϕ be a boolean formula in 3-CNF and $k \in \mathbb{N}$. Let $\mathcal{V}$ be a set of k arbitrarily selected variables in ϕ and $\mathcal{R}$ a set containing the remaining ones. If needed, we fill up $\mathcal{V}$ with dummy variables if there are less than k in ϕ. We then construct an instance of Graph Envy With Pairwise Sharing as follows.

- For every variable $X \in \mathcal{V}$ we introduce three separate nodes V_X, X and $\neg X$. We connect the latter two to the first one and assign resources as shown in the following graph.



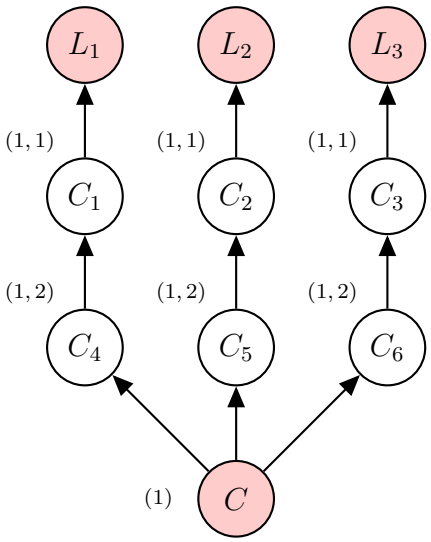
  This leaves both X and $\neg X$ initially envious of V_X.

- For every variable $X \in \mathcal{R}$ we introduce 4 separate nodes D_X, V_X, X and $\neg X$. We connect them and assign resources as shown in the following graph.



  As with the previous configuration, X and $\neg X$ are initially envious of V_X.

- For every clause $C = (L_1 \vee L_2 \vee L_3)$ in ϕ, where the L_i are literals, i.e. either X or $\neg X$ for a variable X, we introduce 7 new nodes C and C_i, $1 \leq i \leq 6$. We denote with L_i the corresponding node introduced through one of the previous constructions (i.e. either X or $\neg X$ for the appropriate variable X). We then connect the new nodes with the literal nodes and assign each of the new nodes resources as shown in the following graph.

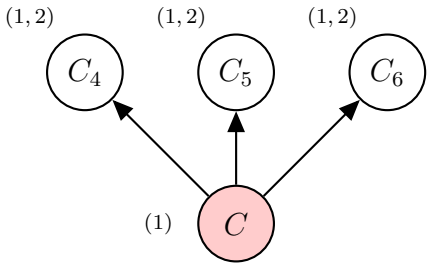


Only C is initially envious (of C_4, C_5 and C_6).

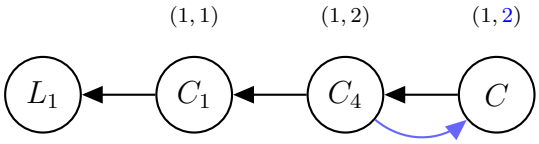
Following this construction, we end up with a graph G consisting of $3\cdot|\mathcal{V}|+4\cdot|\mathcal{R}|+7\cdot|\phi|$ nodes (where $|\phi|$ is the number of clauses in ϕ) and $2\cdot|\mathcal{V}|+4\cdot|\mathcal{R}|+9\cdot|\phi|$ edges and an initial allocation π of $|\mathcal{V}|+2\cdot|\mathcal{R}|+13\cdot|\phi|$ resources on G. We argue that there is a simple 2-sharing allocation extending π with less than or equal to k envious nodes if and only if there is a truth assignment satisfying π.

"⇒": For this direction assume there is a simple 2-sharing allocation $\pi' = (\pi, \pi'_{\mathbf{E}})$ extending π, with less than or equal to k envious nodes. By construction all $2\cdot(\mathcal{V}+\mathcal{R})$ literal nodes are envious in the initial allocation. For the nodes of the variables $X \in \mathcal{V}$, observe that only sharing with V_X could fix them, as the 1-value resources of the C_1, C_2 and C_3 nodes are not sufficient to remove the envy. On the other hand, since only one of the two literal nodes X and $\neg X$ can share with V_X the other must remain envious, leaving exactly $k = |\mathcal{V}|$ envious nodes. From that observation we can conclude that all but those k nodes are not envious in π', and in particular we have $\mathrm{Env}(\pi') \subseteq \mathcal{V}$.

Consider now for a clause C of ϕ the connected nodes C_4, C_5 and C_6.



As C is not envious in π', we know one of them is sharing a 2-value resource with it. W.l.o.g. assume the sharing node is C_4 and consider the path to the literal node L_1.



The node C_1 is also not envious in π'. Now, if L_1 were sharing with V_X, increasing the utility of L_1 to 3, this would leave C_1 envious, unless C_4 would be sharing with it. But since C_4 is already sharing with C, we know L_1 is not sharing with V_X.

This shows that for C not to be envious, it must be connected in the underlying undirected graph to a literal node for a variable X that is not sharing with V_X. We now construct a truth assignment for the variables in ϕ by setting a variable X to true iff the node X is not sharing with V_X. From the reasoning presented above, we then know that every clause contains a literal that is true under that assignment.

"⇐": For the reverse direction assume there is a truth assignment for the variables in ϕ such that the formula ϕ is true. The strategy that follows from the reasoning in the previous part, is to extend the base allocation π by having V_X share with X if the variable X is true under the assignment and with $\neg X$ otherwise. For variables $X \in \mathcal{R}$ we further extend the allocation to have D_X share with the connected literal node that does not share with V_X, resolving that nodes envy. Once again this leaves only one envious literal node for each of the k variables in $\mathcal{V}$.

We now need to show that we can extend the allocation even further to fix the envy of the C_i nodes connected to a literal node and that this still leaves a C_i node that is free to share with C.

For the first part, observe that a node C_i, $1 \leq i \leq 3$, is envious of the connected literal node L if and only if L would be sharing with the connected V_X node, leaving L at a total utility value of 3. To fix the affected C_i node, we extend the allocation to have C_{i+3} share its 2-value resource with it.

For the second part, observe that for each clause C of ϕ there is at least one literal that is true under the truth assignment. By construction, this means the corresponding literal node is not sharing with the connected V_X and the connected C_i thus not envious, even without C_{i+3} lending its 2-value resource. This leaves C_{i+3} free to share with C. Extending the allocation accordingly for all clauses yields a simple 2-sharing allocation with exactly k envious nodes. □

3.2 Parameterized Hardness

The hardness result given in Theorem 3.1 implies that there cannot be an efficient (i.e. polynomial time) algorithm to minimize the number of envious nodes in an allocation by introducing pairwise sharing (unless $P = NP$). The proof shows that this can be seen as direct consequence of the peculiar property of envy to potentially spread when sharings are introduced. As demonstrated by the examples in Section 3.1.1 this can make it hard to envision, how an optimal solution might look like.

We are of course still interested in efficient algorithms computing sharing allocations. While the result from the previous chapter suggests this is likely not possible in the general setup, one might hope that the hardness only manifests for certain instances. In this section we therefore study the hardness of restricted versions of the Graph Envy With Pairwise Sharing problem.

As a first step we can examine the hardness proof for Graph Envy With Pairwise Sharing. In the proof a graph and allocation are constructed from an instance of the 3-Sat problem. This yields already some insights into what is needed for the problem to be hard. In particular, we can see that the constructed graph contains no directed cycles and is indeed a *Directed Acyclic Graph* (DAG). Additionally, no node has a bag of size greater than two in the initial resource allocation. We therefore get the following corollary.

Corollary 3.2. Graph Envy With Pairwise Sharing *is NP-hard even when*

- k *is fixed and*
- G *is a DAG and*
- *the maximum bag-size is at most 2, i.e.* $\max_{i \in V} |\pi(i)| \leq 2$.

The observations presented in the corollary are not the only way to restrict the problem. To formally state the results in this chapter we will use theoretical machinery from the study of *parameterized complexity*. In particular show that for two selected choices of a parameter the problem is not *fixed parameter tractable*, i.e. it cannot be solved in $f(k)n^{o(1)}$ time, where k is the parameter and n is the size of the instance.

In the following two subsections we examine two parameterized versions of the Graph Envy With Pairwise Sharing problem. We show that the respective versions are not *fixed parameter tractable*, i.e. they cannot be solved in $f(k)n^{o(1)}$ time, where k is the parameter and n the size of instance. For a formal introduction to parameterized complexity we refer to the book of Downey and Fellows [DF99].

3.2.1 Number of Sharings

As discussed the hardness of Graph Envy With Pairwise Sharing seems to be related to the possibility of envy being "moved" when sharings are introduced to an

allocation. We demonstrated in the examples in Section 3.1.1 that each sharing introduced to fix a node might make one or more of its neighbors envious. Depending on the graph we might end up with more envious nodes.

One could assume this implies the hardness depends on the number of sharings we perform and if properly restricted could therefore allow us to devise a fast algorithm. More formally, one could hope that Graph Envy With Pairwise Sharing parameterized by the number of sharings that might be added to the initial allocation is *fixed parameter tractable*. The following theorem states that this is not the case.

Theorem 3.3. Graph Envy With Pairwise Sharing *parameterized by the maximum number of sharings in the simple 2-sharing allocations is W[1]-hard.*

Proof. We prove the result by reducing from Independent Set, a well-studied W[1]-hard problem. Let $G = (V, E)$ be a (undirected) graph and $l > 0$. We construct an instance of Graph Envy With Pairwise Sharing.

1. Fix an enumeration of the node in G: $V = \{v_1, \ldots, v_n\}$.
2. Construct a directed graph $G' = (V', E')$, where
   - $V' = V \cup \{p_1, \ldots, p_n\}$
   - $E' = \{(v_i, v_j) \mid \{v_i, v_j\} \in E, i < j\} \cup \{(v_i, p_i) \mid 1 \leq i \leq n\}$
3. Introduce a set of resources $R = \{r_1, \ldots, r_n, s_1, \ldots, s_n\}$
4. Construct an allocation $\pi : V \to 2^R$ of resources R to agents V':
   - $\pi(v_i) := \{r_i\}$
   - $\pi(p_i) := \{s_i\}$
5. Construct a utility function $u : R \to \mathbb{N}$:
   - $u(r_i) := i$
   - $u(s_i) := n + 1$

With this construction we have $u(\pi(v_i)) = i$ and $u(\pi(s_i)) = n+1$. We now know v_i is envious of p_i and any node v_j, $j > i$, it is connected with. The envy towards p_i cannot be fixed by sharing with any of the v_j.

We set $k := n - l$ and argue that (G, l) is a yes-instance of Independent Set if and only if (G', u, π, k) is a yes-instance of Graph Envy With Pairwise Sharing.

Assume (G, l) is a yes instance of Independent Set. W.l.o.g. let then $\{v_1, \ldots, v_l\}$ be an independent set. We construct a simple 2-sharing allocation π' by extending π with

$$p_i \xrightarrow{s_i} v_i \text{ for all } 1 \leq i \leq k.$$

We then have $u(\pi'(v_i)) = i + n + 1$ for $1 \leq i \leq k$. Now let $1 \leq i < j \leq k$ with $(v_i, v_j) \in E'$. Since $\{v_1, \ldots, v_l\}$ is an independent set, we know that $j > l$ and therefore

$$u(\pi'(v_i)) = i + n + 1 > j = u(\pi'(v_j)).$$

We then know that exactly the envy of the nodes in $\{v_1, \ldots, v_l\}$ has been fixed in π', leaving only $n - l = k$ envious nodes.

Now for the reverse direction assume (G', u, π, k) is a yes-instance of Graph Envy With Pairwise Sharing. Let then π' be a simple 2-sharing allocation extending π with $|\operatorname{Env}(\pi')| \leq k$. We set

$$I := V \setminus \operatorname{Env}(\pi').$$

Observe that then $|I| \geq n - k = l$. Now let $1 \leq i < j \leq n$ with $v_i, v_j \in I$. We know $v_i, v_j \notin \operatorname{Env}(\pi')$. As Argued above, this is only possible if p_i and p_j are sharing with them. Assume now that $\{v_i, v_j\} \in E$. Then $(v_i, v_j) \in E'$ by construction and since

$$u(\pi'(v_i)) = n + i + 1 < n + j + 1 = u(\pi'(v_j))$$

this would mean $v_i \in \operatorname{Env}(\pi')$. This is a contradiction, and we therefore know that no such edge can exist and I is an independent set of size at least l in G. □

3.2.2 Number of Envious Nodes in Initial Allocation

Theorem 3.3 indicates that not just the number of necessary sharings to produce an optimal solution determines how hard it is to solve an instance of the Graph Envy With Pairwise Sharing problem. When trying to isolate other means to determine restricted versions of the problem that can be efficiently solved, we therefore want to consider more explicit restrictions to the input.

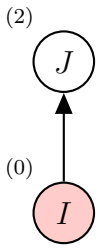
When examining the proofs for both Theorem 3.1 and Theorem 3.3 we can see that in both a graph and initial allocation is constructed. This is of interest, in that this already dictates what nodes are initially envious. In particular there is does not seem to be an upper bound on the number of envious nodes, with half of the nodes introduced in the proof of Theorem 3.3 being initially envious.

One might hope that having only a bounded number of envious nodes — or even just one — might simplify the problem enough to allow the efficient computation of solutions. However, the following theorem states that this is not the case.

Theorem 3.4. Graph Envy With Pairwise Sharing *with $k = 0$ is NP-hard on initial allocations with only one envious node.*

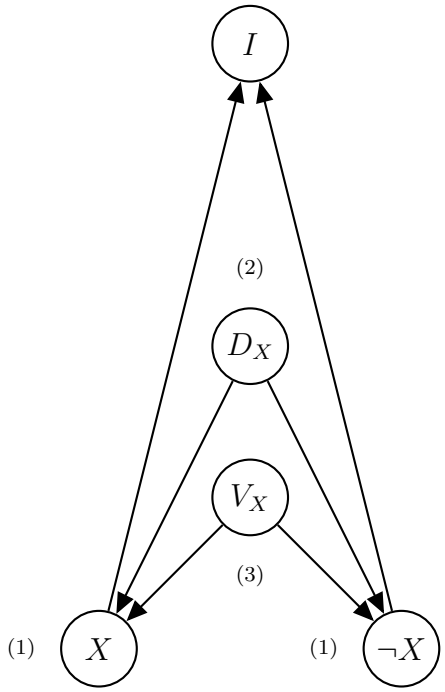
Proof. We present a similar reduction from 3-Sat like the one for the main hardness result in 3.1. Let ϕ be a 3-KNF formula. We construct an instance of Graph Envy With Pairwise Sharing through the following gadgets.

- We add two nodes I and J to the graph and connect them and assign resources as follows.



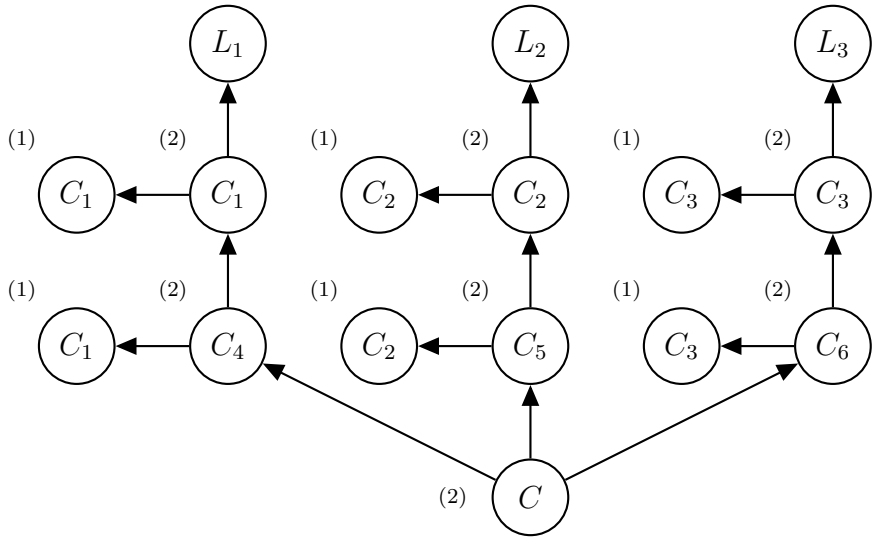
This leaves I envious of J under the initial allocation.

- For each variable X in ϕ we add nodes X, $\neg X$, D_X and V_X to the graph. We then connect them to each other and to E and assign resources with utilities as follows.



We call the nodes X and $\neg X$ the *literal nodes* (for the variable X). Observe that none of the new nodes is initially envious and that only either X or $\neg X$ can share with V_X or D_X.

- For each clause $C = (L_1 \vee L_2 \vee L_3)$ in ϕ we add nodes $C, C_1, \ldots, C_6$, $V_{C_1}, \ldots, V_{C_6}$ to the graph. We then connect them to each other and to the literal nodes L_1, L_2 and L_3 and assign resources as follows.



Again, none of the nodes is initially envious.

Reduction Correctness We now argue that ϕ is a yes-instance of 3-SAT if and only if the constructed instance is a yes-instance of GRAPH ENVY WITH PAIRWISE SHARING with $k = 0$ with I being the only envious node in the initial allocation.

"⇒": For the "⇒" direction assume ϕ is a yes-instance of 3-SAT. We choose a truth assignment that satisfies the formula and start constructing a simple 2-sharing allocation by first having J share with I and then for all variables X have V_X share with X exactly if X is true under the assignment and with $\neg X$ otherwise. The remaining literal node for X should share the resource of D_X. This ensures that all literal nodes are now no longer envious of I.

Now for a clause $C = (L_1 \vee L_2 \vee L_3)$ of ϕ we can see that C_1, C_2 and C_3 are envious of the respective connected literal nodes if and only if that node shares with N_X (where X is the variable corresponding to that literal node). Now if C_1 is envious this is fixed, by making V_{C_1} share its resource with C_1. This in turn leaves C_4 envious, unless similarly fixed by making V_{C_4} share with it. The same construction is done for all pairs of C_2, C_5 and C_3, C_6 nodes. If at least one of the nodes C_4, C_5 or C_6 is sharing this leaves C envious of that node. From the construction so far, we know that C_4 (and similarly C_5 and C_6) is only shared with if the connected literal node is itself connected to the corresponding N_X node. However, since we started from a satisfying truth assignment, we know that at least one of the literal nodes L_1, L_2 or L_3 must be connected to V_X. This leaves the corresponding node C_4, C_5 or C_6 then free to share with C fixing its envy. Then no node is left envious, and we have therefore constructed a simple 2-sharing allocation with no envious nodes.

"⇐" For the reverse direction assume there is a simple 2-sharing allocation π, extending the initial allocation, with $|\operatorname{Env}(\pi)| = 0$. We know that I is envious unless J is sharing with it, which in turn forces X and $\neg X$ to share with V_X or N_X (as sharing with C_1, C_2 or C_3 would make that node envious). We construct a truth assignment by setting a variable X to true if and only if the literal node X is sharing with V_X.

Now assume there is a clause $C = (L_1 \vee L_2 \vee L_3)$ that is not satisfied by the assignment. As we have argued above, this means that all three literal nodes must be connected to the corresponding D_X nodes. However, for π not to leave any envious nodes this means $V_{C_1}, \ldots, V_{C_6}$ must be sharing with $C_1, \ldots, C_6$ respectively. This means the utilities of resources available to C_4, C_5 and C_6 sums up to 4 and C is therefore envious with no one to share with. This contradicts the choice of π as an envy free allocation, meaning there cannot be such a clause. ϕ must then be satisfied by the constructed truth assignment. □

4 Optimal Pairwise Sharing on Paths

In the previous chapter we showed that Graph Envy With Pairwise Sharing is unlikely to be efficiently solvable. The formal result is stated in Theorem 3.1. When analyzing the hardness result we noted that the reduction we used in the proof is build around a rather specific graph structure. In particular, it relies on cycles in the underlying undirected representation of the graph. As a next step in analyzing the hardness of the problem, in this and the following chapter we therefore consider only graphs without cycles.

In particular in this chapter we will restrict the graphs to paths in their underlying representation. We show that there is an — arguably complex — greedy linear-time algorithm. Given an initial allocation this algorithm computes an optimal simple 2-sharing allocation.

Formally, throughout the chapter we denote as G a fixed directed graph

$$G = (\{v_1, \dots, v_n\}, E)$$

with

$$(v_i, v_{i+1}) \in E \oplus (v_{i+1}, v_i) \in E \text{ for all } 1 \leq i < n.$$

The $\oplus$ operator is meant to represent exclusive choice. We know that exactly one of the edges is part of the graph.

Note that the underlying undirected graph is indeed a path going from node v_1 to node v_n in the given order of nodes. The results in this section can trivially be extended to account for graphs consisting of many paths. In addition, we also consider a single utility function u throughout this chapter. This is merely for readability and all results apply for different utility functions $\{u_i\}_{i \in V}$ as well.

4.1 Greedy Sharing Decisions

A key observation motivating this and the next chapter is that in the proofs for both Theorem 3.1 and for Theorem 3.3 we construct graphs that potentially contain undirected cycles. When looking for instances of the Graph Envy With Pairwise Sharing problem we can solve efficiently, it seems therefore appropriate to consider simpler graph classes that do not contain cycles. In this chapter we focus on some of the simplest non-trivial undirected graphs: paths.

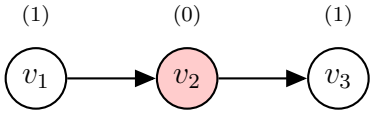
Compared to arbitrary graphs we can immediately see, why paths might be simpler to handle. When introducing a sharing $v_i \to v_j$, we can introduce at most two more envious nodes. By increasing the utility of v_j only v_{j-1} and v_{j+1} can become envious.

And since one of them must be the one sharing with v_j (i.e. $i = j - 1$ or $i = j + 1$) we can only hope to fix the other one by introducing another sharing. This simplifies the reasoning about optimal decisions quite a bit.

This reasoning and the fact that the envy of a node v_i can be fully determined by looking at the bags of the nodes v_i, v_{i-1} and v_{i+1} suggest that it is possible to locally — i.e. by only looking at nodes in a constant distance of v_i — decide if we need to include a sharing involving that node. Employing such a local decision logic then allows the description of a linear-time algorithm that simply considers the nodes in the enumeration order to construct an optimal simple 2-sharing allocation.

Before we present such an algorithm, we want to describe the general ideas and pitfalls of the local decision logic.

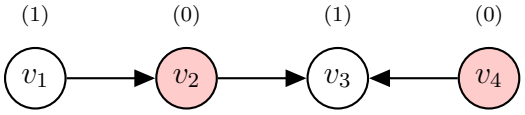
Simple Sharing As a first example consider the following graph consisting of three nodes and some initially assigned resources.



If we want to determine an optimal sharing decision for the envious v_2 it might be tempting to just fix it. In fact, assuming at least one of the outer nodes v_1 or v_3 is free to share, simply adding such sharing to the allocation could fix the envy and thus decrease the number of envious nodes.

An important observation is that any node can only be used to remove envy for one of its neighbors (by sharing one of its resources). In the presented situation, there is therefore no drawback from fixing the envy of v_2, even if this blocks one neighbor from fixing another node, as the total number of envious nodes could then have only been reduced by one in all cases.

There are, however, still some points to consider, when choosing which of the neighbors to share with. Consider the following extension of the example.

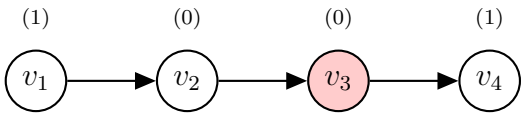


Here, both v_2 and v_4 are envious. The latter can only be fixed by sharing with v_3. This means that even though both, v_1 and v_3, could share their respective resource with v_2 to fix its envy, an optimal solution would always have to use v_1.

With no further knowledge about the allocation this example shows that we would have to consider at least all nodes with distance 2 for any local sharing decision. However, as our algorithm will consider the nodes in order, we can always assume optimal decisions have already been implemented for one of the neighbors. In the example, if the algorithm visits the nodes from left to right, we would thus always prefer the left neighbor and only use the other one if that is not possible.

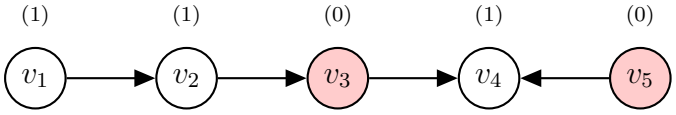
Potential Sharing While the first set of examples gives us a clear indication of when to share if a node is envious, we have yet to come up with rules for sharing with nodes that might not yet be envious.

Consider the following graph as a simple example.



Initially only v_3 is envious. While this can be fixed by having v_4 share with it, v_2 would then be envious in the resulting allocation.

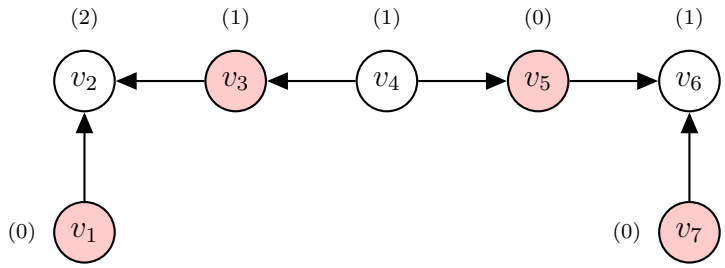
It might be possible to resolve this if v_2 is handled after v_3 one. As we strive, however, for an algorithmic approach visiting the nodes in a fixed order, this might pose a problem. Note that the given example could already be solved by such an algorithm if it introduced some form of preemptive sharing. Even though v_2 is not envious at the time it is handled by the algorithm, the sharing $v_1 \to v_2$ could be implemented. This would basically prepare v_2 for any increase in utility of v_3 at a later step. However, it is not hard to construct examples of similar structure in which such a preemptive sharing would yield a non-optimal solution.



Here, v_2 is not envious. If we followed the suggested approach of preemptively sharing the resource of v_1 with v_2, the envious v_3 could only be fixed by having v_4 share with it. In turn, v_5 would remain envious. We can compare this to the optimal solution, where the second shares with the third and the fourth shares with the last node, leaving no envious nodes.

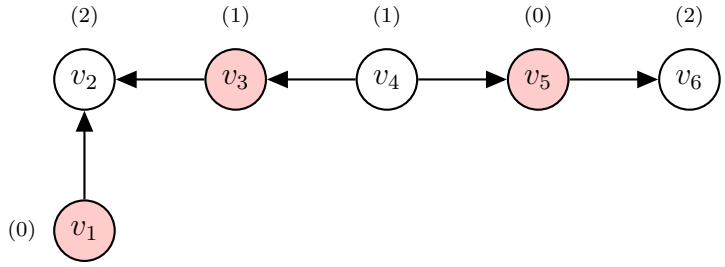
To handle this situation in a more general fashion, a greedy algorithm cannot just include a sharing preemptively. Instead, whenever required by a newly added sharing, it might have to reevaluate an already visited node. It can then introduce another sharing for that node if needed. Both sharings — the one fixing the current node and the one fixing in turn its predecessor — will be added to the current allocation. Note that it is not necessary to look even further back, as it would not be possible to fix envy of the node before the predecessor (since it is the one now sharing its resource). Instead, the predecessor can only increase its utility if this is not creating new envious nodes.

Philanthropic Nodes There is another situation that forces our algorithm to look beyond the neighbors of the current node. Consider the following example.



We again want to include sharings in the enumeration order of the involved nodes. v_1 is envious and v_2 would need to share to fix it. As v_3 is envious of the now occupied v_2 we could consider sharing with v_4, making it envious in the process. However, this would mean that now v_6 can only share with either v_5 or v_7 leaving the other envious. This results in two envious nodes. In contrast, if v_4 does not share with v_3 it is free to share with v_5. Now v_6 can be used to fix v_7 leaving only one envious node in total. This example suggests that when trying to fix v_3 using the next node v_4 and making it envious might not be optimal.

Consider now a slightly modified version of the previous example.



Here, the situation is similar for nodes v_1, v_2 and v_3. This time, we can again consider having v_4 share to fix v_3. Again, this will only leave v_6 to fix v_5 leaving now exactly v_4 envious. We call v_4 a philanthropic node, as it is made envious in the final solution to have both neighbors fixed. If we do not have v_4 share, however, the situation for v_5 does not change, as only v_6 can provide resources with utility value large enough to fix v_5. So even if v_5 is fixed, this now leaves v_3 and v_4 envious, making this solution worse than in the first case.

These exampels show that we cannot decide if we should fix an envious node by making the sharing neighbor envious if we only look at the immediate neighbors. The algorithm presented in the next section instead defers the decision. If it encounters a node that could only be fixed via such a sharing, it will not add this sharing. Instead, if it finds a node that could have its envy fixed by making its predecessor envious, it will check if this predecessor can actually fix *its* predecessor. If so, both sharings are used in the allocation, leaving the predecessor of the originally considered node envious of both its neighbors and making it a philanthropic node.

4.2 The MinPathEnvy Algorithm

Due to the simple structure of paths, it stands to reason that it should be possible to use only the local neighborhood of a node (i.e. only nodes within a constant distance) to decide if and how we can optimally extend a given sharing using that node. As shown in the previous section, such decisions require some careful considerations of several edge cases.

In this section we present a greedy algorithm that given an initial allocation of the resources on the path computes an optimal (i.e. minimal with respect to the number of envious nodes) simple 2-sharing allocation. We then proceed to explain the arguably rather complex technical case distinctions used in the implementation. In the next section we then proceed to prove the correctness and running time of the algorithm.

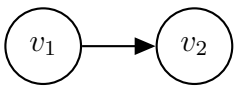
In Figure 4.1 we present the implementation of four helper functions and the scaffolding of the MinPathEnvy algorithm. The first helper function HasRes, spanning lines 1 through 7, can be used to check if the utility of a given node can be increased to be in a given interval by sharing with another node. CanIncrease(π,v_i,v_j,$[a,b]$) returns **true** if and only if a resource of the bag of v_i has big enough utility that, if added to the bag of v_j, increases that utility to be at least a and not bigger than b. We explicitly allow intervals $[a,\infty)$ without upper limit. The next two helper functions MinRes and MaxRes, spanning lines 8 through 15, can be used to select the resource matching the conditions checked by HasRes of minimum or maximum utility respectively. These functions are only meant to be called after it was checked, whether such a resource exists (e.g. by calling HasRes).

The fourth helper function Envious, spanning lines 16 through 22, can be used to check if a node is envious in the given allocation. Envious(π,n) returns **true** if and only if n is envious in π.

Finally, MinPathEnvy, spanning the remaining lines, is the implementation of our algorithm. It consists of one main loop starting in line 25, designed to iterate over all nodes in enumeration order. We omitted the implementation of the loop, as we present different cases for handling a node — along with the code implementing them — in the following subsections.

4.2.1 Case 0: The initial node

This case applies for the first node v_1 and only if there is an arc to the second node v_2, i.e. $(v_1, v_2) \in E$.



The code for this case is given in Figure 4.2. In line 26 we check that it is indeed the first iteration and that v_1 is envious. Note that $v_1 \in \text{Env}(\pi)$ implies $(v_1, v_2) \in E$, as there is no other node v_1 could be envious of. In line 27 it is then checked, whether v_2 contains a resource that can be used to fix the envy of v_1. If that is the case the minimum

```
1: function HasRes(π, s, t, I)
2:     if ∃r ∈ π(s) : u(π(t)) + u(r) ∈ I then
3:         return true
4:     else
5:         return false
6:     end if
7: end function

8: function MinRes(π, s, t, I)
9:     R ← {r ∈ π(s) | u(π(t)) + u(r) ∈ I}
10:    return arg min_{r∈R} u(r)
11: end function

12: function MaxRes(π, s, t, I)
13:    R ← {r ∈ π(s) | u(π(t)) + u(r) ∈ I}
14:    return arg max_{r∈R} u(r)
15: end function

16: function Envious(π, n)
17:    if n ∈ Env(π) then
18:        return true
19:    else
20:        return false
21:    end if
22: end function

23: function MinPathEnvy(π_0)
24:    π ← π_0
25:    for all i ← 1, ..., n do
```

Code for the different cases in this section

```
97:    end for
98:    return π
99: end function
```

Figure 4.1: Listing for the MinPathEnvy algorithm scaffolding

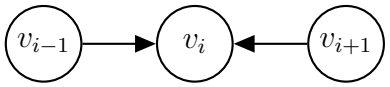
```
26: if i = 1 and v_1 ∈ Env(π) then
27:     if HasRes(π, v_2, v_1, [u(π(v_2)),∞)) then
28:         r ← MinRes(π, v_2, v_1, [u(π(v_2)),∞))
29:         π ← π extended with v_2 -r-> v_1
30:     end if
31: end if
```

Figure 4.2: Code for the *first node* (case 0)

utility resource is selected to be shared with v_1 (lines 28 and 29). It is not necessery but merely an implementation detail to select the resource with minimum utility. In fact, any resource usable to fix the envy of v_1 could be used.

4.2.2 Case 1: A Blind Node

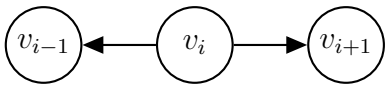
This case applies if the current node v_i does not have any outgoing edges, i.e. if neither $(v_i, v_{i-1}) \in E$ nor $(v_i, v_{i+1}) \in E$.



Without any outgoing edges for v_i, we know that it cannot be envious, regardless of the original allocation $\pi_{\mathbf{N}}$ and any additional utility assigned to either v_{i-1} or v_{i+1} by the algorithm. Strictly speaking, this case does not need to be considered in the implementation.

4.2.3 Case 2: An Observant Node

This case applies if the current node v_i does only have outgoing edges, i.e. if both $(v_i, v_{i+1}) \in E$ and $(v_i, v_{i-1}) \in E$.



We present the implementation for observant nodes in Figure 4.3. Again the code first checks in line 32, whether the edge configuration makes v_i an observant node and whether it is envious. It then proceeds in line 33 to initialize the variable M to contain the maximum utility of the neighbors v_{i-1} and v_{i+1}. To fix the envy of v_i we need to increase its utility to at least M.

We can examine two possible actions to fix the envy of v_i: sharing with v_{i-1} and sharing with v_{i+1}. Ideally, we want to prefer sharing with v_{i-1}, since the algorithm already visited that node.

```
32: if (v_i, v_{i-1}) ∈ E and (v_i, v_{i+1}) ∈ E and Envious(π, v_i) then
33:     M ← max{u(π(v_{i-1})), u(π(v_{i+1}))}
34:     if v_{i-1} ∉_π and HasRes(π, v_{i-1}, v_i, [M,∞)) then
35:         r ← MaxRes(π, v_{i-1}, v_i, [M,∞))
36:         π ← π extended with v_{i-1} -r-> v_i
37:     else if HasRes(π, v_{i+1}, v_i, [M,∞)) then
38:         r ← MinRes(π, v_{i+1}, v_i, [M,∞))
39:         π ← π extended with v_{i+1} -r-> v_i
40:     end if
41: end if
```

Figure 4.3: Code for *observant nodes* (case 2)

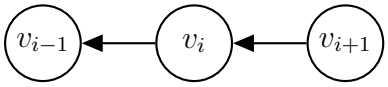
The algorithm therefore first checks in line 34 whether v_{i-1} is free to share (i.e. it is not already sharing) and if its bag contains a resource suitable for fixing the envy of v_i. Here the HasResfunction is used to check whether v_{i-1} has a resource that would increase the utility of v_i above M. Due to the edge configuration sharing any resource with v_i cannot make one of its neighbors envious. The algorithm is therefore free to pick any such resource to share with v_i. When deciding on which resource to use, we need to consider that the algorithm might try to increase the utility of v_{i+1} at a later step. By choosing the maximum utility resource in line 35, the algorithm may later have more freedom to improve the situation for v_{i+1} without reintroducing envy in v_i.

If it is not possible to fix the envy by using a resource from v_{i-1}, the code next checks in line 37, whether v_{i+1} can be used as a sharing partner. As before sharing any of its resources with v_i cannot introduce a new envious node and the algorithm arbitrarily picks the minimum utility resource and adds the sharing to π (in lines 38 and 39).

If none of the neighbors of v_i can provide a suitable resource to fix the envy of v_i, we proceed to the next iteration.

4.2.4 Case 3: A Backward Looking Node

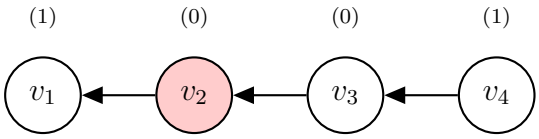
This case applies if the arcs incident to v_i point in reverse direction of the enumeration order of the nodes, i.e. $(v_{i+1}, v_i) \in E$) and $(v_i, v_{i-1}) \in E$.



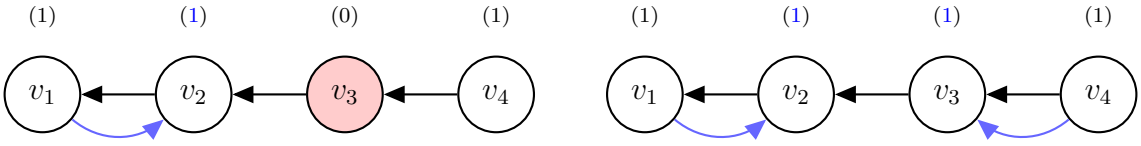
The code for this case is given in Figure 4.4 and first checks that v_i is indeed a backward looking node in line 42.

With v_i envious of v_{i-1} and v_{i+1} potentially being envious of v_i the situation is more complex than for the previous case. To fix the envy of v_i we need to increase its utility. This in turn could introduce envy for v_{i+1}. Ideally, we would want to decrease the

number of envious nodes. However, even exchanging the envy of v_i for the envy of v_{i+1} could be beneficial, as can be seen in the following example.



Here, both v_1 and v_4 each have been allocated a resource with utility value 1, leaving v_2 envious of v_1. The envy can be fixed by having v_1 share its resource with v_2. This in turn would now make v_3 envious of v_2. However, the resource assigned to v_4 can now be shared with v_3 to leave no envious nodes. Thus, by temporarily moving the envy from v_2 to v_3 an optimal solution could be constructed.



If sharing with its predecessor v_{i-1} can fix the envy of v_i, there is never a drawback from doing this even at the expense of making v_{i+1} envious. In this scenario the latter is still free to share and the number of envious nodes does not increase.

Despite this insight, we have to prefer sharings that do not introduce new envious nodes and as before the algorithm should prefer sharing with v_{i-1}, as it has already been visited. In the code this is handled in two different branches. In the first case, handled in line 43, it is checked, whether v_{i+1} is already envious. In this case sharing any resource of v_{i-1} with v_i does not introduce envy. In case such a resource can be used to fix the envy of v_i the algorithm selects the minimum utility resource in line 44 as this may increase the chance of the envy of v_{i+1} being fixed in a later step. The other case assumes v_{i+1} is not already envious. This is checked in line 46 along with whether a resource of v_{i-1} can be used to fix the envy of v_i *without making* v_{i+1} *envious*.

If none of the resources of v_{i-1} matches the criteria the algorithm proceeds to look at the resources of v_{i+1}. Again, there are two branches to account for the fact that v_{i+1} may already be envious. The checks are given in lines 49 and 52 and the corresponding branches mirror the ones introduced for v_{i-1}.

The last branch in line 55 is finally handling the case discussed above. It checks if a resource of v_{i-1} can be used to fix the envy of v_i. Note that it differs from the first two branches in that we now know such a sharing makes v_{i+1} envious. Again, it is important for the algorithm so select the minimum utility resource to increase the chance of fixing the envy of v_{i+1} in a later iteration.

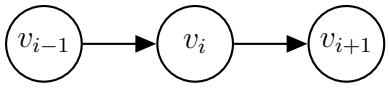
4.2.5 Case 4: A Forward Looking Node

This case applies if the arcs incident to the current node v_i are pointing in the direction of the enumeration order, i.e. $(v_{i-1}, v_i) \in E$ and $(v_i, v_{i+1}) \in E$. As we will see, this case

```
42: if (v_i, v_{i-1}) ∈ E and (v_{i+1}, v_i) ∈ E and Envious(π, v_i) then
43:     if v_{i-1} ⇏_π and Envious(π, v_{i+1}) and HasRes(π, v_{i-1}, v_i, [u(π(v_{i-1})),∞)) then
44:         r ← MinRes(π, v_{i-1}, v_i, [u(π(v_{i-1})),∞))
45:         π ← π extended with v_{i-1} -r-> v_i
46:     else if v_{i-1} ⇏_π and HasRes(π, v_{i-1}, v_i, [u(π(v_{i-1})),u(π(v_{i+1}))]) then
47:         r ← MinRes(π, v_{i-1}, v_i, [u(π(v_{i-1})),u(π(v_{i+1}))])
48:         π ← π extended with v_{i-1} -r-> v_i
49:     else if Envious(π, v_{i+1}) and HasRes(π, v_{i+1}, v_i, [u(π(v_{i-1})),∞)) then
50:         r ← MinRes(π, v_{i+1}, v_i, [u(π(v_{i-1})),∞))
51:         π ← π extended with v_{i-1} -r-> v_i
52:     else if HasRes(π, v_{i+1}, v_i, [u(π(v_{i-1})),u(π(v_{i+1}))]) then
53:         r ← MinRes(π, v_{i+1}, v_i, [u(π(v_{i-1})),u(π(v_{i+1}))])
54:         π ← π extended with v_{i-1} -r-> v_i
55:     else if v_{i-1} ⇏_π and HasRes(π, v_{i-1}, v_i, [u(π(v_{i-1})),∞)) then
56:         r ← MinRes(π, v_{i-1}, v_i, [u(π(v_{i-1})),∞))
57:         π ← π extended with v_{i-1} -r-> v_i
58:     end if
59: end if
```

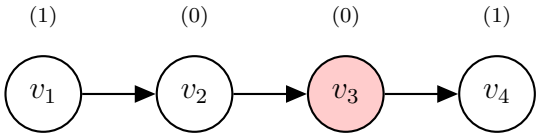
Figure 4.4: Code for *backward looking nodes* (case 3)

is the most involved, as the decision to share can create envy in nodes, the algorithm has already visited before.



As with the other cases we need to be careful not to introduce new envious nodes that leave us in a worse position than before. The situation differs from the previous case, in that now v_{i-1}, a node that has already been visited by the algorithm, would be the one becoming envious. As it might be possible to remedy this situation (by having v_{i-2} share with v_{i-1}), the algorithm effectively has to consider some of the already handled nodes.

To illustrate this situation, we present the following example.

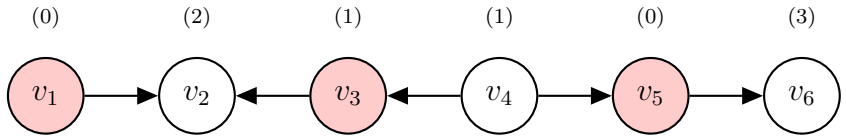


The agorithm can skip v_1 and v_2 as both are not envious. To fix the envy of v_3 it would have to have v_4 share its resource with v_3, making v_2 envious in the process.

Without the algorithm now revisiting v_2 one envious node would remain, even though the addition of v_1 sharing its resource with v_2 would yield an envy-free allocation.

Note that preemptively establishing the sharing $v_{i-2} \rightarrow_\pi v_{i-1}$ during the iteration for v_{i-1} is not possible, as this blocks v_{i-1} from sharing with its successor without reducing the number of envious nodes.

The situation above is not the only case, where the algorithm needs to revisit previous nodes. The following example illustrates another situation, where this becomes necessary.



To fix v_1 the algorithm would have v_2 share with it. This leaves v_4 sharing with v_3 as the only option to fix the former. However, as discussed in the previous case, this extension should not be performed. Instead the algorithm would continue and skip v_4 to then handle the envious v_5. Only v_6 sharing its resource with v_5 fixes the envy, but introduces in turn envy of v_4. It is now not possible to fix v_4. However, looking back even one more step, v_4 sharing its resource with v_3 fixes that nodes envy.

This shows a curious property of this particular setup. When iteratively constructing an optimal solution, it might be necessary to purposefully create envy in nodes that are not envious under the original allocation.

Both these cases leave a rather involved and technical case distinction in the pseudocode. The code for this configuration, given in Figure 4.5 first checks, whether v_i is envious and a forward looking node in line 60, the *potential* p is computed. It is the value by which the utility of v_{i-1} can be increased by sharing with v_{i-2} without making v_{i-2} envious.

Next, similar to the code in the previous case, the algorithm checks if resources from v_{i-1} (lines 70 and 73) or from v_{i+1} (line 76) can be used to fix the envy of v_i without introducing envy in v_{i-1}. In line 79 the algorithm checks if v_{i+1} has a resource that can be used to fix the envy of v_i without increasing the utility of v_i above that of v_{i-1} increased by the *potential* computed before. If this is true the sharing between v_{i+1} and v_i and if needed the sharing between v_{i-2} and v_{i-1} realizing the potential have to be implemented.

The final branch starting in line 86 is concerned with the case that we can make v_{i-1} a philantropic node. If possible, it implements the sharings between v_{i+1} and v_i and between v_{i-1} and v_{i-2} to fix the envy of both v_i and v_{i-2} at the expense of leaving v_{i-1} envious in the final allocation.

60: **if** $(v_{i-1}, v_i) \in E$ and $(v_i, v_{i+1}) \in E$ and $v_i \in \mathrm{Env}(\pi)$ **then**
61: $p \leftarrow 0$
62: **if** $v_{i-1} \not\leftrightarrow_\pi$ **and** $v_{i-2} \not\leftrightarrow_\pi$ **then**
63: **if** $((v_{i-1}, v_{i-2}) \in E$ **or** $\textsc{Envious}(\pi, v_{i-2}))$
64: **and** $\textsc{HasRes}(\pi, v_{i-2}, v_{i-1}, [0,\infty))$ **then**
65: $p \leftarrow u(\textsc{MaxRes}(\pi, v_{i-2}, v_{i-1}, [0,\infty)))$
66: **else if** $\textsc{HasRes}(\pi, v_{i-2}, v_{i-1}, [0,u(\pi(v_{i-2}))])$ **then**
67: $p \leftarrow u(\textsc{MaxRes}(\pi, v_{i-2}, v_{i-1}, [0,u(\pi(v_{i-2}))]))$
68: **end if**
69: **end if**
70: **if** $v_{i-1} \not\leftrightarrow_\pi$ **and** $\textsc{Envious}(\pi, v_{i-1})$ **and** $\textsc{HasRes}(\pi, v_{i-1}, v_i, [u(\pi(v_{i+1})),\infty))$ **then**
71: $r \leftarrow \textsc{MaxRes}(\pi, v_{i-1}, v_i, [u(\pi(v_{i+1})),\infty))$
72: $\pi \leftarrow \pi$ extended with $v_{i-1} \xrightarrow{r} v_i$
73: **else if** $v_{i-1} \not\leftrightarrow_\pi$ **and** $\textsc{HasRes}(\pi, v_{i-1}, v_i, [u(\pi(v_{i+1})),u(\pi(v_{i-1}))])$ **then**
74: $r \leftarrow \textsc{MaxRes}(\pi, v_{i-1}, v_i, [u(\pi(v_{i+1})),u(\pi(v_{i-1}))])$
75: $\pi \leftarrow \pi$ extended with $v_{i-1} \xrightarrow{r} v_i$
76: **else if** $\textsc{Envious}(\pi, v_{i-1})$ **and** $\textsc{HasRes}(\pi, v_{i+1}, v_i, [u(\pi(v_{i+1})),\infty))$ **then**
77: $r \leftarrow \textsc{MinRes}(\pi, v_{i+1}, v_i, [u(\pi(v_{i+1})),\infty))$
78: $\pi \leftarrow \pi$ extended with $v_{i+1} \xrightarrow{r} v_i$
79: **else if** $\textsc{HasRes}(\pi, v_{i+1}, v_i, [u(\pi(v_{i+1})),u(\pi(v_{i-1})) + p])$ **then**
80: $r \leftarrow \textsc{MinRes}(\pi, v_{i+1}, v_i, [u(\pi(v_{i+1})),u(\pi(v_{i-1})) + p])$
81: $\pi \leftarrow \pi$ extended with $v_{i+1} \xrightarrow{r} v_i$
82: **if** $p \neq 0$ **then**
83: $r' \leftarrow \textsc{MinRes}(\pi, v_{i-2}, v_{i-1}, [u(\pi(v_{i-1})) + p,u(\pi(v_{i-1})) + p])$
84: $\pi \leftarrow \pi$ extended with $v_{i-2} \xrightarrow{r'} v_{i-1}$
85: **end if**
86: **else if** $\textsc{HasRes}(\pi, v_{i+1}, v_i, [u(\pi(v_{i+1})),\infty))$
87: **and** $(v_{i-1}, v_{i-2}) \in E$ **and** $(v_{i-2}, v_{i-3}) \in E$
88: **and** $v_{i-1} \not\leftrightarrow_\pi$ **and** $v_{i-2} \not\leftrightarrow_\pi$ **then**
89: $r \leftarrow \textsc{MinRes}(\pi, v_{i+1}, v_i, [u(\pi(v_{i+1})),\infty))$
90: **if** $\textsc{HasRes}(\pi, v_{i-1}, v_{i-2}, [u(\pi(v_{i-3})),\infty))$ **then**
91: $r' \leftarrow \textsc{MinRes}(\pi, v_{i-1}, v_{i-2}, [u(\pi(v_{i-3})),\infty))$
92: $\pi \leftarrow \pi$ extended with $v_{i+1} \xrightarrow{r} v_i$
93: $\pi \leftarrow \pi$ extended with $v_{i-1} \xrightarrow{r'} v_{i-2}$
94: **end if**
95: **end if**
96: **end if**

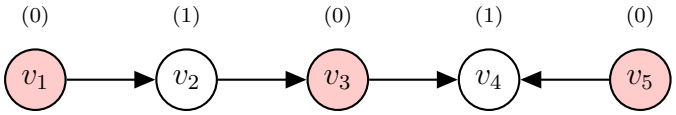
Figure 4.5: Code for *forward looking nodes* (case 4)

4.3 Correctness and Running Time of the MinPathEnvy Algorithm

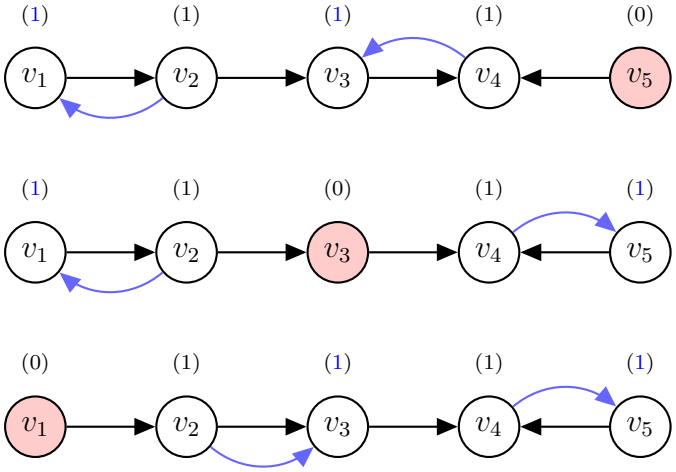
In the previous section we described the MinPathEnvy algorithm and motivated the arguably complex case distinctions necessary to optimally update an allocation. It is by no means obvious the presented formulation of the algorithm is indeed constructing optimal simple 2-sharing allocations from the initial allocations in all cases. In this section we present running time and a formal proof of correctness of the algorithm.

At its core the MinPathEnvy algorithm presented in the previous section iterates once over the nodes and performs locally optimal extension to the allocation. Not only can this be done efficiently in linear time (i.e. linear in the number of nodes), but also suggests it might lend itself well to a proof of correctness via induction. If one is able to show the correctness of the decision for the first node, and can then argue for the correctness and optimality of each extension step, this would yield a complete correctness proof.

However, this strategy is significantly complicated by the fact that there is often not a single unique optimal solution. Instead, there may be several as can be seen from the following small example.



Initially, v_1, v_3 and v_5 are envious. Since only two nodes, v_2 and v_4, can be used to fix the envy, at least one node has to remain envious. With this in mind we can see, however, there are three optimal solutions.



In all three solutions one node is left envious, so all three are optimal with respect to the number of envious nodes. Upon visiting v_1 we cannot easily know whether the decision to have v_2 share with it is optimal. While there are clearly wrong choices the

examples indicate that it might not always be easily possible to argue for any single decision to be optimal.

Instead of just using an inductive line of reasoning we therefore employ a different strategy. Instead of directly arguing for the optimality of the solution, we show that any simple 2-sharing allocation may be updated to adhere to the algorithms local decisions, without increasing the number of envious nodes. We can then use this insight to show that the output of the algorithm cannot produce a higher number of envious nodes than any optimal simple 2-sharing allocation, making it indeed optimal.

Before we present the proof we present for formal main result of this chapter in the following theorem.

Theorem 4.1. *The* MinPathEnvy *algorithm computes a simple 2-sharing allocation that is optimal with respect to the number of envious nodes in* $\mathcal{O}(|V| + |R|)$ *time.*

Before we prove Theorem 4.1 we want to formalize the idea of updating a simple 2-sharing allocation to adhere to the algorithms decision. We do this in two parts. The following lemma states that we can update any allocation that matches the output of the algorithm on all nodes up to an index $i-1$ (i.e. a prefix) to also match on the node v_i. This gives us a characterization of the "greedyness" of the algorithm: if it is possible to fix v_i given the current prefix without introducing unnecessary envious nodes, the algorithm does exactly that.

Lemma 4.2. *Let* $\pi = (\pi_{\boldsymbol{N}}, \pi_{\boldsymbol{E}})$ *be the output of the* MinPathEnvy *algorithm and* $\mu = (\pi_{\boldsymbol{N}}, \mu_{\boldsymbol{E}})$ *another simple 2-sharing allocation. If there is an* $i \in \{1, \dots, n\}$ *with*

$$\operatorname{Env}(\pi) \cap \{v_1, \dots, v_{i-1}\} = \operatorname{Env}(\mu) \cap \{v_1, \dots, v_{i-1}\} \text{ and } v_i \in \operatorname{Env}(\mu) \setminus \operatorname{Env}(\pi)$$

then there is a simple 2-sharing allocation $\nu = (\pi_{\boldsymbol{N}}, \nu_{\boldsymbol{E}})$ *with*

1. $\nu(v_j) = \pi(v_j)$ *for all* $j \leq i$ *and*
2. $v_i \notin \operatorname{Env}(\nu)$ *and*
3. $|\operatorname{Env}(\nu)| \leq |\operatorname{Env}(\mu)|$.

Proof. Let $i \in \{1, \dots, n\}$ with

$$\operatorname{Env}(\pi) \cap \{v_1, \dots, v_{i-1}\} = \operatorname{Env}(\mu) \cap \{v_1, \dots, v_{i-1}\} \text{ and } v_i \in \operatorname{Env}(\mu) \setminus \operatorname{Env}(\pi).$$

This implies

$$|\operatorname{Env}(\pi) \cap \{v_1, \dots, v_i\}| = |\operatorname{Env}(\mu) \cap \{v_1, \dots, v_i\}| - 1.$$

We construct a new simple 2-sharing allocation ν with the desired properties in two steps. We start by defining the simple 2-sharing allocation $\nu_0 = (\pi_{\mathbf{N}}, \nu_{0\mathbf{E}})$ via the following equations for all $j < n$ and $e_j := E \cap \{(v_{j+1}, v_j), (v_j, v_{j+1})\}$.

$$\begin{aligned}
\nu_{0\mathbf{E}}(e_j) &= \pi_{\mathbf{E}}(e_j), \text{ for all } j < i \\
\nu_{0\mathbf{E}}(e_j) &= \emptyset, \text{ for } j \in \{i, i+1\} \\
\nu_{0\mathbf{E}}(e_j) &= \mu_{\mathbf{E}}(e_j), \text{ for all } j \geq i+1
\end{aligned}$$

This construction ensures ν_0 is a valid simple 2-sharing allocation (i.e. no node shares with more than one neighbor and only shares resources it has assigned in $\pi_{\mathbf{N}}$). Additionally, the following properties hold.

- $\nu_0(v_j) = \pi(v_j)$ for all $j < i$
- $|\operatorname{Env}(\nu_0) \cap \{v_1, \ldots, v_{i-1}\}| = |\operatorname{Env}(\pi) \cap \{v_1, \ldots, v_{i-1}\}| = |\operatorname{Env}(\mu) \cap \{v_1, \ldots, v_{i-1}\}|$
- $|\operatorname{Env}(\nu_0) \cap \{v_{i+2}, \ldots, v_n\}| = |\operatorname{Env}(\mu) \cap \{v_{i+2}, \ldots, v_n\}|$

We construct ν by extending ν_0 along the edges e_i and e_{i+1} according to the following cases.

1. If $v_{i+1} \not\rightarrow_\pi v_i$ and $v_{i+1} \xrightarrow{r}_\mu v_{i+2}$ we extend ν_0 with $v_{i+1} \xrightarrow{r} v_{i+2}$. This ensures

   $$v_{i+2} \in \operatorname{Env}(\nu) \Leftrightarrow v_{i+2} \in \operatorname{Env}(\mu).$$

   Since additionally we now have $u(\nu(v_i)) = u(\pi(v_i))$ and $u(\nu(v_{i+1})) = u(\pi(v_{i+1}))$ and therefore $v_i \notin \operatorname{Env}(\nu)$ this yields

   $$\begin{aligned} |\operatorname{Env}(\nu)| \leq & \underbrace{|\operatorname{Env}(\nu) \cap \{v_1, \ldots, v_i\}|}_{=|\operatorname{Env}(\pi)\cap\{v_1,\ldots,v_i\}|=|\operatorname{Env}(\mu)\cap\{v_1,\ldots,v_i\}|-1} + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+1}\}|}_{\leq|\operatorname{Env}(\mu)\cap\{v_{i+1}\}|+1} + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+2}, \ldots, v_n\}|}_{=|\operatorname{Env}(\mu)\cap\{v_{i+2},\ldots,v_n\}|} \\ \leq & |\operatorname{Env}(\mu)| \end{aligned}$$

2. If $v_{i+1} \xrightarrow{r}_\pi v_i$ and $v_{i+1} \not\rightarrow_\mu v_{i+2}$ extend ν_0 with $v_{i+1} \xrightarrow{r} v_i$. This ensures $\nu(v_i) = \pi(v_i)$ and $v_i \notin \operatorname{Env}(\nu)$. Since now $u(\nu(v_{i+1})) = u(\pi_{\mathbf{N}}(v_{i+1})) \leq u(\mu(v_{i+1}))$ and by assumption $u(\nu(v_{i+2})) = u(\mu(v_{i+2}))$ we know

   $$v_{i+2} \in \operatorname{Env}(\nu) \Rightarrow v_{i+2} \in \operatorname{Env}(\mu)$$

   and get for the number of envious nodes

   $$\begin{aligned} |\operatorname{Env}(\nu)| = & \underbrace{|\operatorname{Env}(\nu) \cap \{v_1, \ldots, v_i\}|}_{=|\operatorname{Env}(\pi)\cap\{v_1,\ldots,v_i\}|=|\operatorname{Env}(\mu)\cap\{v_1,\ldots,v_i\}|-1} + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+1}\}|}_{\leq|\operatorname{Env}(\mu)\cap\{v_{i+1}\}|+1} \\ & + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+2}\}|}_{\leq|\operatorname{Env}(\mu)\cap\{v_{i+2}\}|} + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+3}, \ldots, v_n\}|}_{=|\operatorname{Env}(\mu)\cap\{v_{i+3},\ldots,v_n\}|} \\ \leq & |\operatorname{Env}(\mu)| \end{aligned}$$

3. If $v_{i+1} \xrightarrow{r}_\pi v_i$ and $v_{i+1} \rightarrow_\mu v_{i+2}$ extend ν_0 with $v_{i+1} \xrightarrow{r} v_i$. This ensures $\nu(v_i) = \pi(v_i)$ and as in the first case $v_i \notin \operatorname{Env}(\nu)$. To analyze the number of envious nodes in ν we need to distinguish two cases.

a) $v_{i+1} \xrightarrow{r}_{\pi} v_i$ was established in the ith iteration of the algorithm. All the relevant cases do not let v_{i+1} become envious due to this sharing (this is only possible in nd the checks in lines 49 through 54 prevent it). We can therefore conclude that if v_{i+1} is envious in ν, then it is also envious in $\pi_{\mathbf{N}}$. And since $u(\mu(v_{i+1})) = u(\pi_{\mathbf{N}}(v_{i+1}))$ it must also be envious in μ, i.e.

$$v_{i+1} \in \operatorname{Env}(\nu) \Rightarrow v_{i+1} \in \operatorname{Env}(\mu).$$

We then get

$$\begin{aligned} |\operatorname{Env}(\nu)| \leq & \underbrace{|\operatorname{Env}(\nu) \cap \{v_1, \ldots, v_i\}|}_{=|\operatorname{Env}(\pi)\cap\{v_1,\ldots,v_i\}|=|\operatorname{Env}(\mu)\cap\{v_1,\ldots,v_i\}|-1} + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+1}\}|}_{\leq|\operatorname{Env}(\mu)\cap\{v_{i+1}\}|} \\ & + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+2}\}|}_{\leq|\operatorname{Env}(\mu)\cap\{v_{i+2}\}|+1} + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+3}, \ldots, v_n\}|}_{\leq|\operatorname{Env}(\mu)\cap\{v_{i+3},\ldots,v_n\}|} \\ \leq & |\operatorname{Env}(\mu)| \end{aligned}$$

b) $v_{i+1} \xrightarrow{r}_{\pi} v_i$ was instead established in the $(i+2)$th iteration of the algorithm as part of In this case v_{i+1} is a philantropic node, made envious to fix the envy of v_i and v_{i+2} in $\pi_{\mathbf{N}}$. Note that the algorithm only introduces phiolantropic nodes in Case 4: A Forward Looking Node if it is not possible to fix the current node without making its predecessor envious, i.e. if none of cases checked in lines 70, 73, 76 or 79 applies. From this fact and $v_{i+1} \to_{\mu} v_{i+2}$ we get that either v_{i+2} is left envious or v_{i+1} is made envious in μ. Either way we get

$$|\operatorname{Env}(\mu_0) \cap \{v_{i+1}, \ldots, v_n\}| \leq |\operatorname{Env}(\mu) \cap \{v_{i+1}, \ldots, v_n\}| + 1.$$

This then yields

$$\begin{aligned} |\operatorname{Env}(\nu)| \leq & \underbrace{|\operatorname{Env}(\nu) \cap \{v_1, \ldots, v_i\}|}_{=|\operatorname{Env}(\pi)\cap\{v_1,\ldots,v_i\}|=|\operatorname{Env}(\mu)\cap\{v_1,\ldots,v_i\}|-1} + \underbrace{|\operatorname{Env}(\nu) \cap \{v_{i+1}, \ldots, v_n\}|}_{\leq|\operatorname{Env}(\mu)\cap\{v_{i+1},\ldots,v_n\}|+1} \\ \leq & |\operatorname{Env}(\mu)| \end{aligned}$$

In all cases we have ensured $v_i \notin \operatorname{Env}(\nu)$ and $\nu(v_i) = \pi(v_i)$. ν therefore is a simple 2-sharing allocation with the desired properties. □

Lemma 4.2 allows us to update some allocations that only agree with the decisions of the algorithm up to an index i to then agree up to the index $i+1$. Note that this only works for allocations with envy at a position where there is no envy according to the algorithm. We can interpret this in terms of the greedyness: fixing the envy of nodes with a lower index even by risking envy for nodes at higher index does not prevent optimality.

Before we can use this fact to prove the correctness of the main theorem we need to complement the idea of extending prefixes that agree with the algorithm. In the next lemma we show that locally replacing decisions that result in the same envy as those taken by the algorithm does not produce more envious nodes. This illustrates another aspect of the algorithms "greedyness": using the minimum or maximum utility resources in the appropriate cases ensures there are as many optimal options as possible for the following step.

To prove the lemma we need the following observation that can be derived from the main branching decisions in the algorithm. It formalizes the intuition that sharing should remove envy and should not reintroduce envy in already visited nodes.

Observation 4.3. *Let $\pi = (\pi_{\boldsymbol{N}}, \pi_{\boldsymbol{E}})$ be the output of the algorithm. Then no node that is shared with is envious. Formally, we have*

$$v_i \in \mathrm{Env}(\pi) \Rightarrow v_{i-1} \not\rightarrow_\pi v_i \wedge v_{i+1} \not\rightarrow_\pi v_i.$$

We also need the following observation that can be derived from a case-by-case analysis of the algorithm. It states that only necessary sharings, i.e. sharings preventing or removing envy, are added by the algorithm.

Observation 4.4. *Let $\pi = (\pi_{\boldsymbol{N}}, \pi_{\boldsymbol{E}})$ be the output of the algorithm and let $i \leq n$, such that there is a $j \in \{i-1, i+1\}$ with $j \xrightarrow{r}_\pi i$. Then one or both of the following conditions are true.*

1. *We have $(v_i, v_{i-1}) \in E$ and $u(\pi(v_i)) - u(r) < u(\pi(v_{i-1}))$.*
2. *We have $(v_i, v_{i+1}) \in E$ and $u(\pi(v_i)) - u(r) < u(\pi(v_{i+1}))$.*

With these observations we can now present and prove the second utility lemma, needed to prove Theorem 4.1.

Lemma 4.5. *Let $\pi = (\pi_{\boldsymbol{N}}, \pi_{\boldsymbol{E}})$ be the output of the algorithm and let $\mu = (\pi_{\boldsymbol{N}}, \pi_{\boldsymbol{E}})$ be another simple 2-sharing allocation extending the same initial allocation. If there is a $i \in \{1, \ldots, n\}$ such that*

$$\mu(v_j) = \pi(v_j) \text{ for all } j < i \text{ and } v_i \in \mathrm{Env}(\mu) \Leftrightarrow v_i \in \mathrm{Env}(\pi),$$

then there is a simple 2-sharing allocation $\nu = (\pi_{\boldsymbol{N}}, \nu_{\boldsymbol{E}})$ with

$$\nu(v_j) = \pi(v_j) \text{ for all } j \leq i \text{ and } |\mathrm{Env}(\nu)| \leq |\mathrm{Env}(\mu)|.$$

Proof. Let $i \in \{1, \ldots, n\}$ such that

$$\mu(v_j) = \pi(v_j) \text{ for all } j < i \text{ and } v_i \in \mathrm{Env}(\mu) \Leftrightarrow v_i \in \mathrm{Env}(\pi).$$

Similar to the proof of Lemma 4.2 we first construct an intermediate simple 2-sharing allocation $\nu_0 = (\pi_{\mathbf{N}}, \nu_{0\mathbf{E}})$. We define ν_0 via the following equalities for $\nu_{0\mathbf{E}}$ for all $j < n$

and $e_j \in E \cap \{(v_j, v_{j+1}), (v_{j+1}, v_j)\}$.

$$\begin{aligned}\nu_{0\mathbf{E}}(e_j) &= \pi_{\mathbf{E}}(e_j), \text{ for } j < i\\ \nu_{0\mathbf{E}}(e_j) &= \emptyset, \text{ for } j \in \{i, i+1\}\\ \nu_{0\mathbf{E}}(e_j) &= \mu_{\mathbf{E}}(e_j), \text{ for } j > i+1\end{aligned}$$

From this construction we get $\nu_0(v_j) = \pi(v_j)$ for all $j < i$. Additionally we have

$$\begin{aligned}&|\operatorname{Env}(\nu_0) \cap \{v_{i+1}, \dots, v_n\}|\\ &= |\operatorname{Env}(\nu_0) \cap \{v_{i+1}\}| + |\operatorname{Env}(\nu_0) \cap \{v_{i+2}\}| + \underbrace{|\operatorname{Env}(\nu_0) \cap \{v_{i+3}, \dots, v_n\}|}_{=|\operatorname{Env}(\mu)\cap\{v_{i+3},\dots,v_n\}|}\\ &= |\operatorname{Env}(\nu_0) \cap \{v_{i+1}\}| + |\operatorname{Env}(\nu_0) \cap \{v_{i+2}\}| + |\operatorname{Env}(\mu) \cap \{v_{i+2}, \dots, v_n\}|\end{aligned}$$

To obtain ν with the desired properties we therefore need to extend ν_0 in such a way that we have $\nu(v_i) = \pi(v_i)$ and

$$v_{i+1} \in \operatorname{Env}(\nu) \Rightarrow v_{i+1} \in \operatorname{Env}(\mu) \text{ and } v_{i+2} \in \operatorname{Env}(\nu) \Rightarrow v_{i+2} \in \operatorname{Env}(\mu).$$

We now construct ν from ν_0 according to the following cases.

- If $v_i \in \operatorname{Env}(\pi) \cap \operatorname{Env}(\mu)$ then we know from Observation 4.3 that $\not\rightarrow_\pi v_i$. Now since $\mu(v_{i-1}) = \pi(v_{i-1})$ we know that if $v_i \xrightarrow{r}_\pi v_{i-1}$ then also $v_i \rightarrow_r \mu v_{i-1}$. By construction this sharing is also in ν_0 and we construct ν by extending ν_0 with any existing sharing between v_{i+1} and v_{i+2} in μ. As a result we now have $\nu(v_j) = \mu(v_j)$ for all $j \in \{i-1, i, i+1, i+2\}$ and thus even $\nu = \mu$.

  If instead $v_i \rightarrow_\mu v_{i+1}$ we add this sharing to ν_0 to construct ν. In this case and from $\pi(v_{i-1}) = \mu(v_{i-1}$ we also know $v_i \not\rightarrow_\pi v_{i-1}$ and therefore $\nu(v_{i-1}) = \pi(v_{i-1})$. In all other cases we set $\nu := \nu_0$.

  In all cases we get from these additions $\nu(v_{i-1}) = \pi(v_{i-1})$, $\nu(v_i) = \pi(v_i)$ and $v_{i+1} \in \operatorname{Env}(\nu) \Rightarrow v_{i+1} \in \operatorname{Env}(\mu)$ (since $u(\nu(v_i)) \leq u(\mu(v_i))$. This gives us the desired properties.

- If $v_i \notin \operatorname{Env}(\pi) \cup \operatorname{Env}(\mu)$ and $\not\rightarrow_\pi v_i$ we can argue analogously to the previous case.

- If $v_i \notin \operatorname{Env}(\pi) \cup \operatorname{Env}(\mu)$ and $v_{i-1} \rightarrow_\pi v_i$ then this sharing is in ν_0 as well and we construct ν by extending ν_0 with any sharing between v_{i+1} and v_{i+2} in μ. This already gives us $\nu(v_{i-1}) = \pi(v_{i-1}$ and $\nu(v_i) = \pi(v_i)$.

  Now assume $v_{i+1} \in \operatorname{Env}(\nu) \setminus \operatorname{Env}(\mu)$. Since $\nu(v_{i+2}) = \mu(v_{i+2})$, this implies that v_{i+1} is envious of v_i in ν. In particular we have $(v_{i+1}, v_i) \in E$. The sharing $v_{i-1} \rightarrow_\pi v_i$ could then only have been added through Case 3: A Backward Looking Node. Since none of the conditions of the branches in lines 46, 49 or 52 have been true for v_i, we know that it was not possible to fix the envy of v_i without making v_{i+1} envious. However, since $v_i \notin \operatorname{Env}(\mu)$ a sharing to fix the envy of v_i must also have been implemented in μ (increasing the utility of v_i to at least $u(\pi(v_i))$ due

to the code using the minimum utility resource in lines 44 and 56). And from $v_{i+1} \notin \text{Env}(\mu)$ we get that $v_{i+2} \to_\mu v_{i+1}$. By construction, however, we then also have $v_{i+2} \to_\nu v_{i+1}$. Since $\nu(v_{i+2}) = \mu(v_{i+2})$ and $u(\nu(v_i)) = u(\pi(v_i)) \leq u(\mu(v_i))$, this contradicts $v_{i+1} \in \text{Env}(\nu)$. We therefore know

$$v_{i+1} \in \text{Env}(\nu) \Rightarrow v_{i+1} \in \text{Env}(\mu).$$

- If $v_i \notin \text{Env}(\pi) \cup \text{Env}(\mu)$ and $v_{i+1} \to_\pi v_i$ then we add that sharing to ν_0 to construct ν. This again gives us $\nu(v_{i-1}) = \pi(v_{i-1})$ and $\nu(v_i) = \pi(v_i)$.

  Now assume $v_{i+1} \in \text{Env}(\nu) \setminus \text{Env}(\mu)$. We can then distinguish two cases.

  - Assume v_{i+1} is envious of v_i in ν. Then this is also the case in π, and the sharing $v_{i+1} \to_\pi v_i$ could only have been added in Case 4: A Forward Looking Node. This gives us $(v_{i+1}, v_i) \in E$ and therefore $(v_i, v_{i-1}) \in E$ (otherwise v_i could not be envious) and v_{i+1} is a philanthropic node in π. As during the ith iteration none of the branches in Case 3: A Backward Looking Node could be used to fix the envy of v_i, we can conclude from $v_i \notin \text{Env}(\mu)$ and $\mu(v_{i-1}) = \pi(v_{i-1})$ that a similar sharing $v_{i+1} \to_\mu v_i$ exists. This sharing must also result in v_{i+1} being envious of v_i, contradicting the assumption $v_{i+1} \notin \text{Env}(\mu)$.
  - Assume v_{i+1} is not envious of v_i in ν but instead of v_{i+2}. As this is not the case in μ and since we know $\nu(v_{i+2}) = \mu(v_{i+2})$ this means the utility of v_{i+1} must be greater in μ than in ν. Formally, we therefore have both

    $$\nu(v_{i-1}) = \pi(v_{i-1}) = \mu(v_{i-1}) \text{ and } u(\pi(v_{i+1})) = u(\nu(v_{i+1})) < u(\mu(v_{i+1})).$$

    From Observation 4.4 we then get that $v_i \in \text{Env}(\mu)$ contradicting the original assumption. We can now conclude

    $$v_{i+1} \in \text{Env}(\nu) \Rightarrow v_{i+1} \in \text{Env}(\mu).$$

In all cases we could derive $\nu(v_{i-1}) = \pi(v_{i-1}$, $\nu(v_i) = \pi(v_i)$ and

$$v_{i+1} \in \text{Env}(\nu) \Rightarrow v_{i+1} \in \text{Env}(\mu).$$

Observe that additionally in all cases if $i < n - 2$, we get $\nu(v_{i+2}) \subseteq \mu(v_{i+2})$ and from construction we have $\nu(v_{i+3}) = \mu(v_{i+3})$. This gives us

$$v_{i+2} \in \text{Env}(\nu) \Rightarrow v_{i+2} \in \text{Env}(\mu).$$

The simple 2-sharing allocation ν therefore has all the desired properties. □

Lemma 4.5 is constructed to work with Lemma 4.2. We can use the latter to extend prefixes $v_1, \ldots, v_i$ that end with an envious node where none should be according to the algorithm and the former to extend prefixes that do yield the same envy with the exect output of the algorithm. In both cases the changes do not increase the number

of envious nodes. By iteratively applying both lemmas we can therefore argue that the node at the lowest index at which an optimal simple 2-sharing allocation disagrees with the output of the MinPathEnvy algorithm must be envious in the latter.

We use this idea now to proof Theorem 4.1.

Proof of Theorem 4.1. For the running time observe that during the ith iteration of the loop only nodes in

$$\{v_{i-3}, v_{i-2}, v_{i-1}, v_i, v_{i+1}\}$$

and only the resource of these nodes are referenced. This means any single node and any single resource is referenced at most 5 times throughout a run of the algorithm. This means the total running time of MinPathEnvy is $\mathcal{O}(|V| + |R|)$.

For the correctness let $\mu = (\pi_{\mathbf{N}}, \mu_{\mathbf{E}})$ be a simple 2-sharing allocation that is optimal w.r.t. the number of envious nodes.

If $\text{Env}(\pi) = \text{Env}(\mu)$ there is nothing to show. So assume $\text{Env}(\pi) \neq \text{Env}(\mu)$. Let $i \in \{1, \dots, n\}$ be the smallest index such that $v_i \in \text{Env}(\pi) \not\Leftrightarrow v_i \in \text{Env}(\mu)$. From iterating Lemma 4.2 end Lemma 4.5 we get that we can w.l.o.g. assume $\mu(v_j) = \pi(v_j)$ for all $j < i$ and $v_i \in \text{Env}(\pi) \setminus \text{Env}(\mu)$.

There are now two possibilities:

- Assume v_i is envious of v_{i+1} in π. If this was the case during the ith iteration of the algorithm, then neither resources from v_{i-1} nor v_{i+1} could be used to fix it without introducing envy in v_{i-1}. But such a sharing must be part of μ, i.e. v_{i-1} is envious of v_i in μ. This means $i-1$ is the smallest index with $v_i \in \text{Env}(\pi) \not\Leftrightarrow v_i \in \text{Env}(\mu)$. By using the same arguments from above, we may therefore assume w.l.o.g. that v_i was not envious during the ith iteration.

  For v_i to be still envious in the final output, it must then have been made a philanthropic node by uring the $(i+1)$th iteration. From the code we then get $v_i \to_r \pi v_{i-1}$ and v_i is also envious of v_{i-1} in π. But since $\mu(v_{i-1}) = \pi(v_{i-1})$ we then get $v_i \xrightarrow{r}_{\mu} v_{i-1}$ and consequently that v_i is envious of v_{i-1} in μ. This ontradicts $v_i \notin \text{Env}(\mu)$.

- Assume v_i is envious of v_{i-1} in π. Due to $\mu(v_{i-1}) = \pi(v_{i-1})$ we know $v_i \not\leftrightarrow_{\pi} v_{i-1}$. And since the utility of v_{i-1} could not have changed during later iterations, v_i must have been envious during the ith iteration of the algorithm. Then no resources of v_{i-1} could be used to fix it. Additionally, any sharing of a resource of v_{i+1} that could fix v_i would have made v_{i+1} envious. Such a sharing, however, must be part of μ, i.e. $v_{i+1} \to_{\mu} v_i$ and v_{i+1} is envious of v_i in μ. We can now distinguish two cases.
  - If v_{i+1} is not envious of v_{i+2} in μ, then removing the sharing $v_{i+1} \to_{\mu} v_i$ does not increase the number of envious nodes.
  - If v_{i+1} is also envious of v_{i+2} in μ – this implies $v_{i+3} \to_{\mu} v_{i+2}$, as v_{i+1} was not initially emvious – and v_{i+2} is envious, then removing the sharings $v_{i+1} \to_{\mu} v_i$ and $v_{i+3} \to_{\mu} v_{i+2}$ leaves v_i and potentially v_{i+2} envious. It does, however, not increase the number of envious nodes.

– If v_{i+1} is also envious of v_{i+2} in μ and v_{i+2} is not envious, then this makes v_{i+1} a philanthropic node. If the sharing $v_{i+3} \to_\mu v_{i+2}$ fixes the envy of v_{i+2} then the same configuration would have been discovered by the algorithm as part of lines 86-95). If the sharing does not fix the envy of v_{i+2} then removing it and the sharing $v_{i+1} \to_\mu v_i$ again does not increase the number of envious nodes.

In all cases it is possible to construct an optimal simple 2-sharing allocation μ' with $\mu'(v_j) = \pi(v_j)$ for all $j < i$ and $v_i \in \text{Env}(\mu')$.

In both cases we are either able to derive a contradiction or argue for the existance of an optimal simple 2-sharing allocation not matching that case. Either way, we may conclude that there is a simple 2-sharing allocation for which there can be no such index i and we then know $|\text{Env}(\pi)| = |\text{Env}(\mu)|$. This means π is also optimal with respect to the number of envious nodes. □

This concludes our correctness proof of MinPathEnvy. Through Theorem 4.1 we have now established that Graph Envy With Pairwise Sharing can be solved efficiently on paths (or collection of paths). This is contrasting the hardness results from the previous chapter and motivates the examination of more complex — and arguably more practically relevent — graphs. In the next chapter we therefore look at the class of *trees*.

5 Optimal Pairwise Sharing on Trees

From Theorem 3.1 we know that Graph Envy With Pairwise Sharing cannot be solved efficiently for all instances. And even though there is an inherent connection between simple 2-sharing allocations and matchings on graphs — we have used this to devise fast algorithms for optimizing the social welfare in Chapter 2 — there are reasons to assume the problem of minimizing the number of envious nodes remains hard even under many restrictions. In the previous chapter, however, we presented an algorithm that is capable of optimally extending an allocation on paths in linear time. Consequently, it is now of interest to determine the hardness of the problem on less restricted graph classes.

While the result stated in Theorem 4.1 is interesting, as it shows that the hardness of Graph Envy With Pairwise Sharing is connected to the graph structure, it might also be too limiting for many real-world applications. One of the motivating examples for graph-envy and also simple 2-sharing allocations are management hierarchies in companies. For many cases models with underlying undirected paths will not suffice to describe such structures.

In this chapter we therefore want to look at another simple class of undirected graphs: trees. In a tree all nodes are connected with each other through paths without there being any cycles. We want trees to be rooted, i.e. there is a node marked as the *root*. We first present some arguments complicating the construction of a greedy algorithm, like the one we devised for paths. We then proceed to describe an algorithm based on dynamic programming, which computes the optimal extension to a given allocation on graphs with underlying trees in polynomial time.

Throughout this chapter let $G = (V, E)$ be a directed graph, whose underlying undirected graph is a tree rooted in $\text{root} \in V$. We write

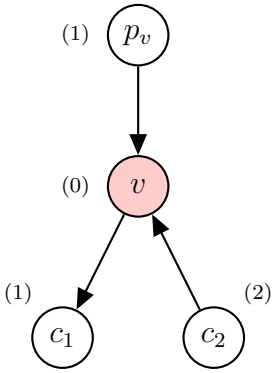
- $p(i) \in V$ for the parent of $i \in V \setminus \{\text{root}\}$,
- $C(i) \subseteq V$ for the children of $i \in V$,
- $C_{\text{in}}(i) := \{j \in C(i) \mid (j, i) \in E\}$ for the children of $i \in V$ connected to i via an incoming arc $(j, i) \in E$ and
- $C_{\text{out}}(i) := \{j \in C(i) \mid (i, j) \in E\}$ for the children of $i \in V$ connected to i via an outgoing arc $(i, j) \in E$.

We call $i \in V$ a leaf node if $C(i) = \emptyset$.

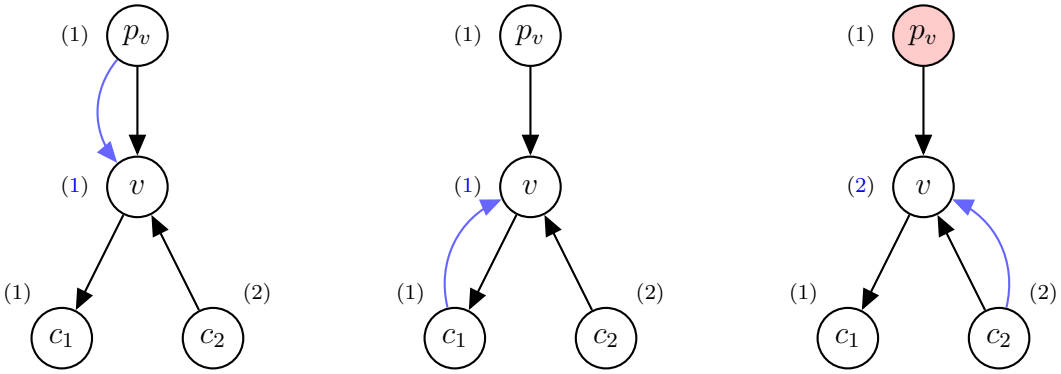
For simplicity, we will refer to a single utility function u throughout this chapter. This is merely syntactical convenience and all results in this chapter apply for different utility functions $\{u_i\}_{i \in V}$.

5.1 The Problem With Greedy Sharing Decisions

Starting from the algorithm for paths described in the previous chapter, one might hope to find a similar reformulation for trees. And in many cases the necessary case distinctions seem to generalize the ideas underlying the MinPathTree algorithm. Consider the following depiction of an inner node v of a tree and the connection to its parent $p_v := p(v)$ and the two children c_1 and c_2.

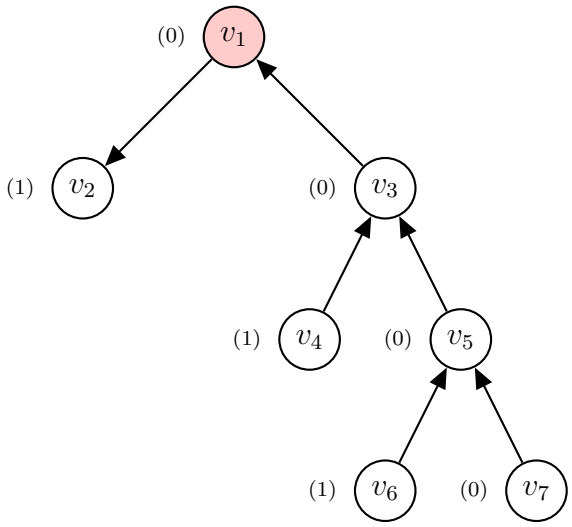


Initially v is envious of c_1. Depending on the current simple 2-sharing allocation, any of the nodes connected to v could share with it to fix the envy. This would result in one of the following three configurations.



In all versions v is no longer envious. However, when sharing the resource of c_2, as seen in the last configuration, p_v is now envious of v. This example shows that, as with paths in the previous section, in trees one would need to consider several distinct cases when deciding on which resource to share with a node. In fact, there are some similarities between the setup shown in the example and the considerations for *backward looking nodes* presented in Section 4.2.4. As we did in that section, we can here as well argue that it may always be correct to "move" the envy from v to p_v if necessary. In a later step the algorithm may than fix the envy of p_v by having $p(p_v)$ or one of the children of p_v share with it.

Despite the similarities for some sharing decisions in trees and paths, there is one difference complicating greedy decisions. In Section 4.1 we describe *potential sharing*, i.e. the need to consider sharings involving two already visited nodes. While necessary to construct an optimal simple 2-sharing allocation, this step only requires to look at a constant number of nodes. Even when implementing the sharing, no other nodes already visited by the algorithm are affected. For trees this is no longer the case. Consider the following example that is mostly identical to the one we used in 3.1.1 to demonstrate how envy can "spread" through a graph.



Initially the only envious node in this example is v_1. This can be remedied by sharing the resource of v_2 with it. This in turn would make v_3 envious. Notice that now the situation for v_3 is identical to how it was for v_1 before any sharings. Repeating the same logic, we could have v_4 share with v_3 and then v_6 with v_5. In this case we would leave v_7 (and potentially more children) envious, indicating that the first sharing between v_2 and v_1 should not have been added. Without v_7, however, following the steps above would yield a simple 2-sharing allocation without any envious nodes.

This example suggests that it is not easily possible to decide if a sharing is part of an optimal solution by just looking at a constant number of nodes.

5.2 A Dynamic Programming Algorithm

The example in the previous section illustrates on the one hand, how the problem of optimizing an allocation with respect to the number of envious nodes is more complex on trees than it is on paths. And while we cannot rule out the possibility of there being a greedy algorithm for trees, such an algorithm would need to be even more complex to reason about, as it would need to account for situations like in the example above.

On the other hand the example also gives us a starting point for a different approach. Consider a setting where, when considering a sharing that would create envy, we already

knew if this envy could be removed by introducing other sharings. Clearly, this would solve the problem demonstrated in the previous section. Such an information would have to be computed as a separate result from the sharing and made available to the algorithm. Following this idea ultimately leads us to dynamic programming, one of the algorithmic standard techniques on trees.

In this section we present a dynamic programming algorithm, capable of determining the number of envious nodes in an optimal simple 2-sharing allocation extending a given allocation in polynomial time. For the remainder of this chapter, we assume there is a fixed initial allocation π_0 on G.

Our algorithm is constructed around the table T, used to store the information, whether it is possible to construct a simple 2-sharing allocation with less than n envious nodes.

$$\begin{aligned} T[i, r, n, e] \in \{\top, \bot\}, \text{ where} \\ i \in V, \\ r \in \bar{R} := R \cup \{\text{non}\}, \\ n \in \mathbb{N}, \text{ and} \\ e \in \{\text{envy}, \text{noenvy}\}. \end{aligned}$$

Intuitively, we want $T[i, r, n, e]$ to be $\top$ if there is a simple 2-sharing allocation π extending π_0 with at most n envious nodes. In π the node i should only be envious *of one of its children* if $e = \text{envy}$ and should share the resource r (or not share if $r = \text{non}$). The r argument is necessary for two reasons. Firstly, it indicates whether i is sharing in π or whether can be used in a new sharing. Secondly, the parameter also allows to compute the final utility of i in π, which can then be used to determine if other nodes are envious of i. We also store the information, whether i is envious of one of its children in the parameter e for a similar reason. To compute the number of envious nodes, we need to understand if by making i envious of its parent, the algorithm is actually introducing envy for i.

Formally, we need to ensure that $T[i, r, n, e] = \top$ if and only if there is a simple 2-sharing allocation π extending π_0 such that

- $r = \text{non}$ or $p(i) \xrightarrow{r}_{\pi'} i$ or there is a child $j \in C(i)$ with either $i \xrightarrow{r}_{\pi'} j$ or $j \xrightarrow{r}_{\pi'} i$,
- $|\operatorname{Env}(\pi') \cap T(i)| \leq n'$, where
  - $T(i)$ is the subtree rooted in i and
  - $n' = n$ if i is not envious of its parent in π or $e = \text{envy}$, and $n' = n - 1$ otherwise, and
- $e = \text{envy}$ or for all $j \in C_{\text{out}}(i)$ we have $u(\pi(i)) \geq u(\pi(j))$.

Given these conditions for the entries of T we can then compute the minimal number of envious node in a simple 2-sharing allocation extending π_0 as

$$\min \left\{ n \in \mathbb{N} \middle| \exists r \in \bar{R} : (T[\text{root}, r, n, \text{envy}] = \top) \vee (T[\text{root}, r, n, \text{noenvy}] = \top) \right\}.$$

In the following subsections we now proceed to describe, how the individual entries of T can be computed. The process will be split. We first define some base cases, for which the computation does not rely on other entries. After that we show how the remaining entries can be computed recursively, i.e. by relying on other entries that have been computed at an earlier step.

5.2.1 Computing the Outer Table

For many combinations of the parameters of T we can compute the value without relying on other entries. For example, there are some cases that cannot yield a valid simple 2-sharing allocation. Most notably, while we allow r to be any resource for simplicity, only those usable in a sharing with the active node i are considered in computations. This is formalized in the first condition presented for entries containing $\top$ presented above. We therefore set $T[i, r, n, e] = \bot$ for $i \in V$, $r \in R \cup \{\text{non}\}$, $n \in \mathbb{N}$ and $e \in \{\text{envy}, \text{noenvy}\}$ if

$$r \notin \{\text{non}\} \cup \pi(i) \cup \pi(p(i)) \cup \bigcup_{j \in C(i)} \pi(j).$$

The other type of entries that can be easily computed without relying on other entries are those for leaf nodes. For each leaf node $i \in V$, $r \in \pi(p(i)) \cup \{\text{non}\}$, $n \in \mathbb{N}$ and $e \in \{\text{envy}, \text{noenvy}\}$ we simply set

$$T[i, r, n, e] = \top.$$

This is due to the fact that we only consider envy towards the children of the selected node when comparing with n. Since i is a leaf node, every simple 2-sharing allocation satisfies the constraints.

To determine the value of T for non-leaf nodes, we need to use the information computed for the children of these nodes. This restricts the order in which we can handle the nodes. One possibility is to handle the nodes *layer-wise*. A *layer* consists of all nodes of equal distance from the root. By handling those with a greater distance before those with a lower distance, we ensure that table entries for all children of a node have been computed before we look the node itself.

With this idea in mind, we can now focus on computing the entries of T for any non-leaf node $i \in V$. Recall that we want any entry $T[i, r, n, e]$ to be $\top$ if there is a simple 2-sharing allocation with at most n envious nodes in the sub-tree rooted in i. While we need to consider the potential envy of i itself, the rest of the at least $n - 1$ envious nodes can be distributed over all the sub-trees rooted in any of the children of i. To compute the entries for i it will therefore be necessary to consider all possibilities to spread the envious nodes over the $m := |C(i)|$ sub-trees. This problem is the well-known n-multicombination problem, and we therefore know that there are at least $\binom{m+n-2}{n}$ such combinations. As this number is not polynomially bounded in the size of the instance, simply iterating over and checking the combinations would not result in an efficient algorithm.

To work around this when computing the entry $T[i, r, n, e]$, we use another table A, containing the information if we can split the envious nodes over the sub-trees rooted in any of the children of i. Let $c_1, \ldots, c_m \in C(i)$ be an arbitrary but fixed enumeration of i's children. Then the table A is given as

$$\begin{aligned} A[j, s, n', e] \in \{\top, \bot\}, \text{ where} \\ j \in \{1, \ldots, m\}, \\ s \in \{\text{share}, \text{noshare}\}, \\ n' \in \{1, \ldots, n\}, \text{ and} \\ e' \in \{\text{envy}, \text{noenvy}\}. \end{aligned}$$

We explain in the next section, how A can be computed efficiently. For now, we just assume there are the following already computed entries.

- $A[m, \text{share}, n, \text{noenvy}]$: This entry is $\top$ if and only if there is a simple 2-sharing allocation π' such that there are n envious nodes in all the sub-trees $T(c_1), \ldots, T(c_2)$. Additionally, i is not envious of one of the children in π' and the sharing indicated by the parameter r (of T) is included in π'.

- $A[m, \text{share}, n-1, \text{envy}]$: This entry is $\top$ if and only if there is a simple 2-sharing allocation π' such that there are $n-1$ envious nodes in all the sub-trees $T(c_1), \ldots, T(c_2)$. Additionally, i is envious of at least one of its children in π' and the sharing indicated by the parameter r (of T) is included in π'.

Given these entries, the entry in T can then be computed. We set $T[i, r, n, e] = \top$ if

$$\begin{aligned} (e = \text{noenvy} \wedge A[m, \text{share}, n, \text{noenvy}] = \top) \\ \vee (e = \text{envy} \wedge A[m, \text{share}, n-1, \text{envy}] = \top) \end{aligned}$$

and $T[i, r, n, e] = \bot$ otherwise.

5.2.2 Computing The Inner Table

As discussed above it is not efficient to iterate over all the at least $\binom{m+n-2}{n}$ ways to split the envious nodes over the $m := |C(i)|$ sub-trees rooted in the children of a node i. To work around this we want to use an *inner* dynamic programming algorithm computing the result of different allocations of the envious nodes to the sub-trees. Note that this algorithm needs to be executed for each entry in T.

Let $i \in V$ a non-leaf node, $r \in \bar{R}$, $n \in \mathbb{N}$ and $e \in \{\text{envy}, \text{noenvy}\}$ such that $T[i, r, n, e]$ is not set to $\bot$ as part of the initialization. Let $c_1, \ldots, c_m \in C(i)$ be an arbitrary but fixed enumeration of the children of i. We then build the table A described through

$$
\begin{aligned}
A[j, s, n', e] \in \{\top, \bot\}, \text{ where} \\
j \in \{0, \ldots, m\}, \\
s \in \{\text{share}, \text{noshare}\}, \\
n' \in \{1, \ldots, n\}, \text{ and} \\
e' \in \{\text{envy}, \text{noenvy}\}.
\end{aligned}
$$

We want $A[j, s, n', e] = \top$ if and only if there is a simple 2-sharing allocation, such that the total number of envious nodes in the sub-trees rooted in the children $c_1, \ldots, c_j$ is at most n' and if $s = \text{share}$ and $r \in \pi(i)$ there is a $k \leq j$ such that i is sharing r with c_k. The parameter e' indicates, whether i becomes envious under that allocation.

Again, there are some cases for which we can determine the entry of A without relying on other entries. In particular for $j = 0$ the value of A does not depend on any actual allocation (as there are not children taken into account). We can distinguish the following four cases.

- $A[0, \text{noshare}, n', \text{noenvy}] = \top$
- $A[0, \text{noshare}, n', \text{envy}] = \bot$
- $A[0, \text{share}, n', \text{noenvy}] = \bot$
- $A[0, \text{share}, n', \text{envy}] = \bot$

These entries are meant to work with the logic for the recursive computation presented in the remaining subsection. Intuitively, the last three cases cannot yield $\top$, since they do not guarantee either sharing with i (as would be required with $s = \text{noshare}$) or the envy of i (as would be required with $e = \text{envy}$).

After performing the initialization, we can then proceed to compute the remaining entries. Each entry $j > 1$ only depends on entries for $j-1$, so it is sufficient to compute A in the order of increasing j values. Let now $j > 1$ and $n' \in \{1, \ldots, n\}$. Several properties of the graph and the allocation π influence the values in A. Table 5.1 contains conditions that need to hold for specific configurations of the parameters. We set $A[j, s, n', e'] = \top$ if the corresponding condition in the table is true and $A[j, s, n', e'] = \bot$ otherwise.

To simplify the visual presentation of the conditions we use some shorthand notation in the table.

- We use $\_$ as a generic placeholder and write $T[i, r, n, \_] = \top$ when we do not care about the envy state of i. Formally, the expression should be read as if it expands to
$$(T[i, r, n, \text{envy}] = \top \vee T[i, r, n, \text{noenvy}] = \top)$$
- We use the indicator function χ to only consider the utility of resources shared in a particular direction. In the table we write $\chi^c(j, r') := \chi_{\pi(c_j)^c}(r')$ to denote a coefficient that is 1 if r' is *not* in $\pi(c_j)$ and 0 otherwise.

While Table 5.1 itself is rather technical, the general structure of the conditions always consists of three elements. In addition to the simplifications shown above, we also use colors to mark the different elements.

We highlight in blue the recursive check whether the siblings support the configuration. The conditions always refer to entries for $j-1$ siblings. Additionally, we allow up to $n-n'$ envious nodes in the sub-trees of the siblings, since n' envious nodes should be the limit for envious nodes in the sub-tree for c_j. For $s = \text{share}$ and/or $e' = \text{envy}$ we need to check if the implied condition (is sharing a resource from i and/or is making i envious) is fulfilled by the children. If not, we need to ensure that c_j is fulfilling it.

- For $s = \text{noshare}$ and $e' = \text{envy}$ we need to check if the children up to c_{j-1} already ensure the envy of i or if this needs to be achieved by c_j:

$$(\ A[j-1, \text{noshare}, n-n', \text{envy}] = \top\ ) \vee (\ A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top\ )$$

- For $s = \text{share}$ and $e' = \text{noenvy}$ we similarly need to check if the children up to c_{j-1} are already using the shared resource of i or if we need to use it to share with c_j:

$$(\ A[j-1, \text{share}, n-n', \text{noenvy}] = \top\ ) \vee (\ A[j-1, \text{share}, n-n', \text{noenvy}] = \top\ )$$

- For $s = \text{share}$ and $e' = \text{envy}$ we need to combine the cases of the previous configurations. This results in 4 different cases to be considered:

$$\begin{aligned}
&(\ A[j-1, \text{share}, n-n', \text{envy}] = \top\ )\\
\vee&(\ A[j-1, \text{noshare}, n-n', \text{envy}] = \top\ )\\
\vee&(\ A[j-1, \text{share}, n-n', \text{noenvy}] = \top\ )\\
\vee&(\ A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top\ )
\end{aligned}$$

We highlight in green the recursive check whether the sub-tree rooted in c_j supports the configuration with n' envious nodes. The conditions are generally similar but change slightly depending on the used resource and whether c_j is envious of i. This is due to the fact that in T we do not consider envy towards the parent of the selected node. In particular the condition is presented in one of the following forms.

- In case i is already sharing (either with its parent or another child) we need to consider all possible sharings between c_j and its children. This includes the possibility to not share (i.e. $r' = \text{non}$).

$$\exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top$$

  In this notation we do not care about the envy of c_j. If there is an arc from c_j to i we also need to consider that c_j is now envious of i.

$$\exists r' \in \bar{R} \setminus \pi(i) : (\ T[c_j, r', n'-1, \text{noenvy}] = \top\ ) \vee (\ T[c_j, r', n', \text{envy}] = \top\ )$$

In the first case there is a simple 2-sharing allocation that leaves c_j not envious. With c_j now becoming envious of i we need to add it to the count of envious nodes. This means we cannot allow more than $n' - 1$ envious nodes in the sub-tree. If c_j is already envious of one or more of its children, there is no need to decrement the allowed number of envious nodes.

- If c_j is sharing a resource with i, we need to check for suitable allocations that do not have c_j involved in any sharing in the sub-tree.

$$T[c_j, \text{non}, n', \_] = \top$$

This case also requires a similar distinction of the envy of c_j as in the first case, i.e. if c_j is envious of i we need to lower the amount of allowed envious nodes for the lookup.

- Finally, if $s = \text{share}$ we need to consider the case of c_j being the node i is sharing with.

$$T[c_j, r, n', \_] = \top$$

In case of an arc from c_j to i the distinction regarding the envy of c_j is similar to the first case, i.e. if c_j is envious of i we need to lower the amount of allowed envious nodes for the lookup.

We highlight in red the check whether either i or c_j would become envious under the considered sharing. We need to take into account the utility value of the bags of both nodes under the initial allocation π and the added utility of the resources shared. The actual condition can take several forms depending on the considered sharings.

In the most general case, both i and c_j might be on the receiving side of a sharing. In particular if r is a resource initially assigned to one of the other children we aim to find a resource for c_j.

$$\exists r' \in \bar{R} \setminus \pi(i) : u(\pi(i)) + u(r) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r')$$

Depending on the case, either $u(r)$ or $u(r')$ might disappear from the condition and the relation operator is different. We use the indicator function $\chi^c(j, r')$ to make sure we only add the utility of resources that are not initially assigned to it (i.e. that it receives from one of its children and not giving away). Note that in the case of $r' = \text{non}$ we just define the term $\chi^c(j, r') \cdot u(r')$ to be 0.

5.3 Result

The dynamic programming algorithm presented in the previous sections, though rather technical, does not involve any complex computations. Instead, tables T and A can be computed efficiently. This leads us to the following theorem as the main result of this chapter.

$s = \text{noshare}, e' = \text{noenvy}$	
$r \in \pi(i), (i, c_j) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) \geq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r')$
$r \in \pi(i), (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top))$
$r \in \pi(c_j), (i, c_j) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge T[c_j, \text{non}, n', \_] = \top \wedge u(\pi(i)) + u(r) \geq u(\pi(c_j))$
$r \in \pi(c_j), (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge (u(\pi(i)) + u(r) \leq u(\pi(c_j)) \wedge T[c_j, \text{non}, n', \_] = \top)$ $\vee (u(\pi(i)) + u(r) > u(\pi(c_j)) \wedge (T[c_j, \text{non}, n'-1, \text{noenvy}] = \top \vee T[c_j, \text{non}, n', \text{envy}] = \top))$
$r \in \pi(c_k), (i, c_j) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) + u(r) \geq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r')$
$r \in \pi(c_k), (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) + u(r) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) + u(r) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top))$
$r = \text{non}, (i, c_j) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) \geq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r')$
$r = \text{non}, (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top))$
$s = \text{noshare}, e' = \text{envy}$	
$r \in \pi(i), (i, c_j) \in E$	$(A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top)$ $\vee (A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r'))$
$r \in \pi(i), (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top))$
$r \in \pi(c_j), (i, c_j) \in E$	$(A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge T[c_j, \text{non}, n', \_] = \top)$ $\vee (A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge T[c_j, \text{non}, n', \_] = \top \wedge u(\pi(i)) + u(r) < u(\pi(c_j)))$
$r \in \pi(c_j), (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge (u(\pi(i)) + u(r) \leq u(\pi(c_j)) \wedge T[c_j, \text{non}, n', \_] = \top)$ $\vee (u(\pi(i)) + u(r) > u(\pi(c_j)) \wedge (T[c_j, \text{non}, n'-1, \text{noenvy}] = \top \vee T[c_j, \text{non}, n', \text{envy}] = \top))$
$r \in \pi(c_k), (i, c_j) \in E$	$(A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top)$ $\vee (A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) + u(r) < u(\pi(c_j)) + \chi^c(j, r') \cdot u(r'))$
$r \in \pi(c_k), (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) + u(r) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) + u(r) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top))$
$r = \text{non}, (i, c_j) \in E$	$(A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge \exists r' \in' barR \setminus \pi(i) : T[c_j, r', n', \_] = \top)$ $\vee (A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) \geq u(\pi(c_j)) + \chi^c(j, r') \cdot \cdot u(r'))$
$r = \text{non}, (c_j, i) \in E$	$A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top))$
$s = \text{share}, e' = \text{noenvy}$	
$r \in \pi(i), (i, c_j) \in E$	$(A[j-1, \text{share}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) \geq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r'))$ $\vee (A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge T[c_j, r, n', \_] = \top \wedge u(\pi(i)) \geq u(\pi(c_j)) + u(r))$
$r \in \pi(i), (c_j, i) \in E$	$(A[j-1, \text{share}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top)))$ $\vee (A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge (u(\pi(i)) \leq u(\pi(c_j)) + u(r) \wedge T[c_j, r, n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + u(r) \wedge (T[c_j, r, n'-1, \text{noenvy}] = \top \vee T[c_j, r, n', \text{envy}] = \top)))$
$r \in \pi(c_j), (i, c_j) \in E$	*same as for* $s = \text{noshare}, e' = \text{noenvy}$
$r \in \pi(c_j), (c_j, i) \in E$	
$r \in \pi(c_k), (i, c_j) \in E$	
$r \in \pi(c_k), (c_j, i) \in E$	
$r = \text{non}, (i, c_j) \in E$	
$r = \text{non}, (c_j, i) \in E$	
$s = \text{share}, e' = \text{envy}$	
$r \in \pi(i), (i, c_j) \in E$	$(A[j-1, \text{share}, n-n', \text{envy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top)$ $\vee (A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge T[c_j, r, n', \_] = \top)$ $\vee (A[j-1, \text{share}, n-n', \text{noenvy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : T[c_j, r', n', \_] = \top \wedge u(\pi(i)) \geq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r'))$ $\vee (A[j-1, \text{noshare}, n-n', \text{noenvy}] = \top \wedge T[c_j, r, n', \_] = \top \wedge u(\pi(i)) < u(\pi(c_j)) + u(r))$
$r \in \pi(i), (c_j, i) \in E$	$(A[j-1, \text{share}, n-n', \text{envy}] = \top \wedge \exists r' \in \bar{R} \setminus \pi(i) : (u(\pi(i)) \leq u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge T[c_j, r', n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + \chi^c(j, r') \cdot u(r') \wedge (T[c_j, r', n'-1, \text{noenvy}] = \top \vee T[c_j, r', n', \text{envy}] = \top)))$ $\vee (A[j-1, \text{noshare}, n-n', \text{envy}] = \top \wedge ((u(\pi(i)) \leq u(\pi(c_j)) + u(r) \wedge T[c_j, r, n', \_] = \top)$ $\vee (u(\pi(i)) > u(\pi(c_j)) + u(r) \wedge (T[c_j, r, n'-1, \text{noenvy}] = \top \vee T[c_j, r, n', \text{envy}] = \top))))$
$r \in \pi(c_j), (i, c_j) \in E$	*same as for* $s = \text{noshare}, e' = \text{envy}$
$r \in \pi(c_j), (c_j, i) \in E$	
$r \in \pi(c_k), (i, c_j) \in E$	
$r \in \pi(c_k), (c_j, i) \in E$	
$r = \text{non}, (i, c_j) \in E$	
$r = \text{non}, (c_j, i) \in E$	

Table 5.1: The cases relevant for computing the table A.

Theorem 5.1. Graph Envy With Pairwise Sharing *can be solved on trees in* $\mathcal{O}(|V|^4 \cdot |R|^2)$ *time.*

Proof. We argue that the algorithm presented in the previous sections can be used to solve Graph Envy With Pairwise Sharing.

For the running time of the algorithm, observe that there cannot be more than $|V|$ envious nodes and we therefore only access entries for $n \leq |V|$. This means table T contains $\mathcal{O}(|V|^2 \cdot |R|)$ relevant entries. For table A, we can see that since $|C(i)| < |V|$ for all nodes i there are $\mathcal{O}(|V|^2)$ entries. Now, while computing an entry for T from A (only the entries $A[|C(i)|, \text{share}, n, \text{noenvy}]$ and $A[|C(i)|, \text{noshare}, n-1, \text{envy}]$ are checked) can be done in constant time, the maximum number of operations needed to compute one of the conditions presented in Table 5.1 is $\mathcal{O}(|R|)$. Putting this all together, we can conclude that T can be build up in $\mathcal{O}(|V|^4 \cdot |R|^2)$ time. This is then also the time needed for the whole algorithm, as extracting the optimal result can be done in $\mathcal{O}(|V| \cdot |R|)$.

Correctness of the algorithm follows from construction of the tables T and A. From the interpretation of the table T we can see that finding the smallest n for which we can find a $\top$ entry for the root is already giving us the minimum number of envious nodes we can get when extending the initial allocations to a simple 2-sharing allocation. The validity of the interpretation of T stems from the validity of the interpretation of A.

Correctness of the interpretation for entries in A similarly stems from construction. Due to Table 5.1 we consider all possible cases to extend simple 2-sharing allocations optimal on sub-trees rooted in children of the selected node i. The entries in Table 5.1 describe the optimal scenario for extending the allocation under the induction hypothesis that both the interpretation of A and the interpretation of T are correct for already computed entries. □

Theorem 5.1 shows that Graph Envy With Pairwise Sharing can be solved efficiently on graphs with trees as their underlying undirected graph. The running time presented, however, makes this a mostly theoretical result. For practical applications the presented algorithm will often be too slow. This is contrasting the results for paths, where Theorem 4.1 suggests our algorithm may be fast enough for real-world instances.

However, the presented upper-bound for the complexity of solving Graph Envy With Pairwise Sharing is likely not optimal and future versions of the algorithm or even novel approaches may very well show far better results.

6 Conclusion

Many applications of resource allocation algorithms — such as housing allocations or the fair distribution of rewards to employees — are inherently linked to social structures. Students in two-person dorm rooms are certainly affected by the choice of their roommate (see the work of Dusselier et al. [Dus+05]). Consequently, the use of social networks to determine the optimum — and especially fairness — of allocation problems suggests itself as a field of future research.

In this chapter, we summarize and discuss the contributions to this field presented in this thesis. We then provide an outlook on future work related to our results.

6.1 Discussion & Future Work

Building upon the concept of *graph-envy*, as given by Bredereck et al. Abebe et al. Chevaleyre et al. and others, we present an approach to improving a given resource allocation through pairwise sharing [AKP17; BKN18; CEM17]. While arguably simple, our model is motivated by real-world applications of the resource allocation theory and already yields some interesting hardness results. It may be interesting to derive similar results for other sharing models (e.g. sharing between more than two agents, sharing more than one resource, ...) and compare them to our findings.

Our model is closely related to *matchings* on the underlying graph. We use this insight to devise efficient algorithms to optimize two different versions of social welfare: utilitarian social welfare and egalitarian social welfare. The possibility to efficiently optimize these values is not surprising, considering that much of the hardness discussed in later parts of the thesis seems to be closely connected to the graphs being directed. Arc directions, however, do not play any role in the social welfare computations we consider.

Looking at a more involved metric, we examine the minimization of *envy* along the social network. We introduce the NP-hard Graph Envy With Pairwise Sharing problem and present some parameterized hardness results. Interestingly enough, some "obvious" restrictions do not affect the hardness. It seems that it is not trivial to find parameters to be used for the restriction of hardness. The main problem, namely envy being "spread", is a rather intuitive effect of sharing and is likely to complicate the problem for other models of sharing as well.

Aiming at further characterizing the hardness of Graph Envy With Pairwise Sharing, we present the MinPathEnvy algorithm capable of solving the problem on paths in linear time and prove its correctness. The algorithm contains several rather technical case distinctions but the underlying concept is straightforward. The result is

interesting as it demonstrates the existence of graph classes on which the problem can be solved efficiently.

Building on this idea, we present a dynamic programming algorithm that solves the Graph Envy With Pairwise Sharing problem on trees in polynomial time. The details are again rather technical and there are undoubtedly many improvements one could think of, especially regarding running time. The results are still promising and suggest there might be more graph classes the problem can be solved on efficiently. Potential candidates may be graphs of bounded tree-width. It might also be interesting to investigate the influence of other graph-related parameters on the hardness of Graph Envy With Pairwise Sharing, such as the number of feedback edges.

Literature

[AKP17] R. Abebe, J. Kleinberg, and D. C. Parkes. "Fair division via social comparison". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2017, pp. 281–289 (cit. on pp. 18, 29, 75).

[AS99] A. Abdulkadiroğlu and T. Sönmez. "House allocation with existing tenants". In: *Journal of Economic Theory* 88.2 (1999), pp. 233–260 (cit. on pp. 9, 10).

[Ast84] A. W. Astin. "Student involvement: A developmental theory for higher education". In: *Journal of college student personnel* 25.4 (1984), pp. 297–308 (cit. on p. 10).

[BKN18] R. Bredereck, A. Kaczmarczyk, and R. Niedermeier. "Envy-Free Allocations Respecting Social Networks". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. Stockholm, Sweden: International Foundation for Autonomous Agents and Multiagent Systems, 2018, pp. 283–291 (cit. on pp. 9, 10, 18, 29, 75).

[Bou+16] S. Bouveret, Y. Chevaleyre, N. Maudet, and H. Moulin. "Fair Allocation of Indivisible Goods". In: *Handbook of Computational Social Choice*. Ed. by F. Brandt, V. Conitzer, U. Endriss, J. Lang, and A. D. Procaccia. Cambridge University Press, 2016, 284–310 (cit. on p. 11).

[CEM17] Y. Chevaleyre, U. Endriss, and N. Maudet. "Distributed fair allocation of indivisible goods". In: *Artificial Intelligence* 242 (2017), pp. 1–22 (cit. on pp. 18, 29, 75).

[Che+06] Y. Chevaleyre, P. E. Dunne, U. Endriss, J. Lang, M. Lemaître, N. Maudet, J. Padget, S. Phelps, J. A. Rodríguez-Aguilar, and P. Sousa. "Issues in Multiagent Resource Allocation". In: *Informatica* 30 (2006), pp. 3–31 (cit. on pp. 18, 21, 24, 29).

[Dam+15] A. Damamme, A. Beynier, Y. Chevaleyre, and N. Maudet. "The power of swap deals in distributed resource allocation". In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2015, pp. 625–633 (cit. on p. 11).

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. 1st ed. Springer-Verlag New York, 1999 (cit. on p. 36).

[Dus+05] L. Dusselier, B. Dunn, Y. Wang, M. C. Shelley, and D. F. Whalen. "Personal, Health, Academic, and Environmental Predictors of Stress for Residence Hall Students". In: *Journal of American College Health* 54.1 (2005), pp. 15–24 (cit. on pp. 10, 29, 75).

[Edm65] J. Edmonds. "Paths, trees, and flowers". In: *Canadian Journal of mathematics* 17 (1965), pp. 449–467 (cit. on p. 23).

[Fes54] L. Festinger. "A theory of social comparison processes". In: *Human relations* 7.2 (1954), pp. 117–140 (cit. on pp. 10, 29).

[Gab18] H. N. Gabow. "Data Structures for Weighted Matching and Extensions to B-matching and F-factors". In: *ACM Trans. Algorithms* 14.3 (June 2018), 39:1–39:80 (cit. on p. 23).

[GLW17] L. Gourvès, J. Lesca, and A. Wilczynski. "Object Allocation via Swaps along a Social Network". In: *26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. Melbourne, Australia, Aug. 2017, pp. 213–219 (cit. on p. 11).

[HZ79] A. Hylland and R. Zeckhauser. "The Efficient Allocation of Individuals to Positions". In: *Journal of Political Economy* 87.2 (1979), pp. 293–314 (cit. on p. 9).

[Kol09] V. Kolmogorov. "Blossom V: a new implementation of a minimum cost perfect matching algorithm". In: *Mathematical Programming Computation* 1.1 (2009), pp. 43–67 (cit. on p. 23).

[KT06] J. Kleinberg and É. Tardos. *Algorithm design*. International edition. Boston, MA: Pearson, 2006 (cit. on p. 32).

[Mar17] E. Markakis. "Approximation Algorithms and Hardness Results for Fair Division with Indivisible Goods". In: *Trends in Computational Social Choice*. Ed. by U. Endriss. AI Access, 2017. Chap. 12, 231–347 (cit. on p. 11).

[MV80] S. Micali and V. V. Vazirani. "An $O(\sqrt{|V|} \cdot |E|)$ algoithm for finding maximum matching in general graphs". In: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. 1980, pp. 17–27 (cit. on p. 27).

[Pro13] A. D. Procaccia. "Cake cutting: not just child's play." In: *Commun. ACM* 56.7 (2013), pp. 78–87 (cit. on p. 11).

[Pro15] A. D. Procaccia. "Cake Cutting Algorithms". In: *Handbook of Computational Social Choice, chapter 13*. Cambridge University Press, 2015 (cit. on p. 11).

[SS74] L. Shapley and H. Scarf. "On cores and indivisibility". In: *Journal of mathematical economics* 1.1 (1974), pp. 23–37 (cit. on p. 11).

[Ste48] H. Steinhaus. "The Problem of Fair Division". In: *Econometrica* 16.1 (1948), pp. 101–104 (cit. on p. 11).