Approximation Algorithms

for

Submodular Maximization and Network Design Problems

Gilad Kutiel
Approximation Algorithms

for

Submodular Maximization and Network Design Problems

Research thesis

Submitted In Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Gilad Kutiel

Submitted to the Senate of the Technion — Israel Institute of Technology
Adar II, 5779 Haifa
Haifa
March 2019
This research was carried out under the supervision of Prof. Roy Schwartz and Dr. Dror Rawitz.

Publication List:


Acknowledgements

I would like to thank my advisors Prof. Roy Schwartz and Dr. Dror Rawitz. This thesis wouldn’t have been completed without your professional guidance and support. It was a pleasure to work with you, I have learned a lot from you (both on Computer Science and in general). You were always there to help me, teach me and motivate me.

I would like to thank my co-authors, It was a pleasure to work with you and I learned a lot from you.

Finally, I would like to thank my family, Talia, Yoav and Yael for accompanying me throughout my entire PhD.
# Contents

1 Introduction ........................................... 3

2 Convex Recoloring ........................................ 6
   2.1 Introduction ......................................... 6
   2.2 Hardness Result ....................................... 8
   2.3 Properties of Optimal Recolorings ................. 9
      2.3.1 Path-Recoloring ................................ 9
      2.3.2 Path Independence ............................... 11
   2.4 Greedy Algorithm ................................... 13
      2.4.1 The Algorithm .................................. 13
      2.4.2 Analysis for General Graphs .................. 13
      2.4.3 Analysis for Paths .............................. 18
   2.5 Parameterized Complexity ........................... 19

3 Service Chain Placement in SDNs ....................... 20
   3.1 Introduction ......................................... 20
      3.1.1 Related Work .................................. 20
      3.1.2 Our Results .................................... 22
   3.2 Preliminaries ........................................ 23
      3.2.1 Model ............................................ 23
      3.2.2 Notation ........................................ 25
   3.3 Hardness Results ..................................... 25
   3.4 Algorithms for Physical Directed Acyclic Graphs ... 26
      3.4.1 Placing a Sub-chain in a Physical Node ....... 27
      3.4.2 Placing a Service Chain ......................... 27
      3.4.3 Placing a Service Chain with a Latency Bound 28
      3.4.4 FPTAS for General Costs ....................... 29
   3.5 General Networks ................................... 30
      3.5.1 Randomized Algorithm .......................... 30
      3.5.2 Fixed Parameter Algorithm ..................... 32
   3.6 Fault Tolerance ....................................... 32
      3.6.1 Layered Graph Transformation .................. 34
      3.6.2 Fault Tolerance on Layered Graphs ............. 34
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Convex Recoloring - Example of convex and non-convex colorings of a graph</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Convex Recoloring - The 2-Convex Recoloring problem</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Convex Recoloring - terminology</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Convex Recoloring - Example of retains-all and non-retains-all (optimal) convex recolorings</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Convex Recoloring - Terminology</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Convex Recoloring - Greedy versus optimal solutions</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>Convex Recoloring - Approximation ratio as function of $\phi$</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>Convex Recoloring - Tight analysis</td>
<td>17</td>
</tr>
<tr>
<td>2.9</td>
<td>Convex Recoloring - Tight analysis for paths</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>SCP - Example input and output</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>SCP - Hardness (single node)</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>SCP - Hardness (paths)</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>SCP - Dynamic programming illustration</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>SCP - Orientation phase</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>SCP - Block graph example</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>SCP - Greedy algorithm, worst case</td>
<td>34</td>
</tr>
<tr>
<td>3.8</td>
<td>SCP - Layered graph transformation</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Carpool - Local search improvement</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>Carpool - Example of a star graph</td>
<td>60</td>
</tr>
<tr>
<td>5.7</td>
<td>Carpool - A tight example for Corollary 4</td>
<td>62</td>
</tr>
<tr>
<td>5.8</td>
<td>Carpool - Not submodular</td>
<td>63</td>
</tr>
</tbody>
</table>
Abstract

In this thesis we develop and analyze approximation algorithms, while focusing on two families of problems: network design and submodular optimization. For the former family we consider the problems of Convex Recoloring and the Service Chain Placement in SDNs. For the latter family we consider the problems of maximizing a monotone submodular function given a knapsack constraint and the Maximum Carpool Matching problem.

In a typical instance of a network design problem we are given a collection of resources and our goal is to construct a desired network that satisfies some requirements while utilizing the minimum possible amount of resources.

In the Convex Recoloring problem, we are given a set $C$ and a coloring of an undirected graph, $G(V, E)$, is a function $\chi : V \rightarrow C$. We say that a coloring is convex if the vertices of each color induce a connected graph of $G$. The goal is to find a convex coloring of the graph that recolors the minimum number of vertices. In this thesis we consider a special case of this problem, namely the 2-Convex Recoloring problem, in which the original coloring uses each color in $C$ to color at most two vertices. For this special case we develop a greedy $3/2$-approximation algorithm. We show that if the input graph is a path then this is in fact a $5/4$-approximation algorithm.

In the Service Chain Placement in SDNs problem our goal is to find an optimal resource allocation in a software defined networks that support network function virtualization. We model the problem and give a fully polynomial time approximation scheme for the special case where the physical network is a directed acyclic graph. For general graphs we give a parameterized algorithm. This algorithm finds an optimal solution efficiently for cactus network.

Submodularity is a fundamental mathematical notion that captures the concept of economy of scale and is prevalent in many areas of science and technology. Given a ground set $U$ a set function $f : 2^U \rightarrow \mathbb{R}_+$ over $U$ is called submodular if it has the diminishing returns property: $f(A \cup \{a\}) - f(A) \geq f(B \cup \{a\}) - f(B)$ for every $A \subseteq B \subseteq U$ and $a \in U \setminus B$. Submodular functions naturally arise in different disciplines such as combinatorics, graph theory, probability, game theory, and economics.

In this thesis we consider the problem of maximizing a monotone, submodular function given a Knapsack constraint and develop a fast and simple approximation algorithm for this problem. We also consider the Maximum Carpool Matching problem, where we seek to find an optimal way in which a group of people should share their ride. We show that this problem can be formalized as a non-monotone, unconstrained, submodular function maximization, thus admits a $1/2$-approximation.
Notation

\( G \) — Graph
\( \mathcal{C} \) — Set of colors
\( \chi \) — Coloring function
\( G_\chi \) — Colored Graph
\( \mathcal{G} \) — Virtual graph
\( p \) — Processing capacity/demand
\( b \) — Bandwidth capacity/demand
\( c \) — Cost function
\( f \) — Submodular function
\( M \) — Carpool matching
\( \mathcal{P}_M \) — Passengers w.r.t. \( M \)
\( \mathcal{D}_M \) — Drivers w.r.t. \( M \)
\( \mathcal{Z}_M \) — Solo drivers w.r.t. \( M \)
\( deg_{\text{in}}^M \) — Out degree w.r.t. \( M \)
\( deg_{\text{out}}^M \) — In degree w.r.t. \( M \)
Chapter 1

Introduction

In this thesis we develop and analyze approximation algorithms, while focusing on two families of problems: network design and submodular optimization. For the former family we consider the problems of Convex Recoloring and the Service Chain Placement in SDNs. For the latter family we consider the problems of maximizing a monotone submodular function given a knapsack constraint and the Maximum Carpool Matching problem. Let us now elaborate on the above.

Network Design Problems

In a typical instance of a network design problem we are given a collection of resources and our goal is to construct a desired network that satisfies some requirements while utilizing the minimum possible amount of resources. One of the classical network design problems (and probably the most well known) is the Minimum Spanning Tree problem where, given an undirected weighted graph, our goal is to find the minimum cost set of edges that induces a connected graph. Network design problems have numerous applications, e.g., the construction of efficient communication and traffic networks, the design of small and cheap VLSI chips and the reconstruction of phylogenetic trees, are just a few.

Many of the practical network design problems are NP-hard and thus likely intractable. In this thesis we focus on approximation algorithms for two network design problems which we now briefly (and somewhat informally) describe.\footnote{A formal, detailed, description of each problem will be given at the relevant chapters.}

Convex Recoloring

In Chapter 2 we study the Convex Recoloring problem. Let $C$ be a set of colors, a coloring of an undirected graph, $G(V, E)$, is a function $\chi : V \rightarrow C$. We say that a coloring is convex if the vertices of each color induce a connected sub-graph of $G$. In the Convex Recoloring problem we are given an undirected graph $G$ and a (non-convex) coloring of this graph and our goal is to find a convex coloring of the graph that recolors the minimum number of vertices. In this thesis we consider a special case of this problem, namely the 2-Convex Recoloring problem, in which the original coloring uses each color in $C$ to color at most two vertices. For this special case we develop a greedy $3/2$-approximation algorithm. We show that if the input graph is a path then this is in fact a $5/4$-approximation algorithm.
Service Chain Placement in SDNs

Traditional communication networks use ad-hoc controllers to achieve various functionality and to provide various services for the clients of the network. An emerging paradigm aims to construct networks of general, smart controllers that can be programmed to achieve the same goal. This new paradigm offers much more flexibility to the network administrator that can now construct many virtual networks on top of the physical one. The question that arises is how to allocate resource efficiently in order to satisfy clients’ demands. In Chapter 3 we model the problem and, using scaling and non-trivial dynamic programming techniques, we give a fully polynomial time approximation scheme for the special case where the physical network is a directed acyclic graph. For general graphs we give a parameterized algorithm. This algorithm finds an optimal solution efficiently for cactus network.

Submodular Function Maximization

Submodularity is a fundamental mathematical notion that captures the concept of economy of scale and is prevalent in many areas of science and technology. Given a ground set \( U \) a set function \( f : 2^U \to \mathbb{R}_+ \) over \( U \) is called submodular if it has the diminishing returns property: \( f(A \cup \{a\}) - f(A) \geq f(B \cup \{a\}) - f(B) \) for every \( A \subseteq B \subseteq U \) and \( a \in U \setminus B \). Submodular functions naturally arise in different disciplines such as combinatorics, graph theory, probability, game theory, and economics. Some well-known examples include coverage functions, cuts in graphs and hypergraphs, matroid rank functions, entropy, and budget additive functions. Additionally, submodular functions play a major role in many real-world applications, e.g., the spread of influence in networks [66, 67, 68, 92], recommender systems [33, 34], document summarization [32, 87, 88], and information gathering [53, 76, 77, 78, 80], are just a few such examples.

Combinatorial optimization problems with a submodular objective have been the focus of intense research in the last decade as such problems provide a unifying framework that captures many fundamental problems in the theory of algorithms and numerous real-world practical applications. Examples of the former include, e.g., Max-CUT and Max-DiCUT [42, 51, 54, 59, 63, 69, 107], Max-k-Coverage [43, 109, 108], Max-Bisection [12, 48, 55, 110], Generalized-Assignment [26, 29, 46, 44], and Max-Facility-Location [6, 31, 30], whereas examples of the latter include, e.g., pollution detection [79], gang violence reduction [104], outbreak detection in networks [84], exemplar based clustering [52], image segmentation [71], and recommendation diversification [111].

A main driving force behind the above research is the need for algorithms that not only provide provable approximation guarantees, but are also fast and simple to implement in practice. This need stems from the sheer scale of the applicability of submodular maximization problems in diverse disciplines, and is further amplified by the fact that many of the practical applications arise in areas such as machine learning and data mining where massive data sets and inputs are ubiquitous.

In this thesis we consider maximizing a monotone submodular function given a Knapsack constant, we also consider the Maximum Carpool Matching optimization problem and show

---

2 An equivalent definition is: \( f(A) + f(B) \geq f(A \cup B) + f(A \cup B) \) for every \( A, B \subseteq U \).

3 Many of the above mentioned problems can also be found in introductory books to approximation algorithms [109, 108].

4 Refer to the recent book [13] and survey [74] for additional examples and applications of submodularity in machine learning.
that this problem can be formulated as a submodular function maximization problem. We now briefly describe these problems.

**Submodular Knapsack**

In Chapter 4 we consider the problem of maximizing a monotone, submodular function given a knapsack constraint. There is a well known $(1 - e^{-1})$-approximation algorithm for this problem that runs in $O(n^5)$ time. This is the best approximation ratio that can be achieved by a polynomial time algorithm [70]. A (asymptotic) faster $1 - e^{-1} - \epsilon$-approximation algorithm is also known. This algorithm runs in $1/\epsilon O(1/\epsilon^4) n \log^2 n$ time for any choice of $\epsilon$ and, as the authors of this algorithm mention, is impractical. In this thesis we develop a framework that for any $\alpha < \beta < 1/2$ and an $\alpha$-approximation algorithm $A$ with a running time $t(n)$ produces a $\beta$-approximation algorithm $B$. The running time of $B$ is $O(n^2) + O(t(n))$ where the constants hidden by the big $O$ depend on $\alpha$ and $\beta$. This framework can be used, for example, to develop a $(1 - e^{-2/3})$-approximation algorithm that runs in time $O(n^2)$ with very small constant.

**Maximum Carpool Matching**

In Chapter 5 we consider the Maximum Carpool Matching problem. In this problem we are given an edge weighted directed graph, $G = (V, E)$, and capacity constraints on the vertices, $c : V \rightarrow \mathbb{N}$. Our goal is to find a matching, $M \subseteq E$, with maximum total weight such that in $H = (V, M)$ the out degree of each vertex is at most one, the in degree of each vertex, $v$, is at most $c(v)$, and either the out degree or the in degree of each vertex is zero. We develop approximation algorithms for several special cases of the problem. We also show that MAXIMUM CARPOOL MATCHING can be formalized as a non-monotone, unconstrained, submodular function maximization and, thus, admits a 1/2-approximation algorithm. Finally we develop a $(1/2 - \epsilon)$-approximation algorithm for a more general problem that captures MAXIMUM CARPOOL MATCHING.

---

5 A formal, detailed, description of each problem will be given at the relevant chapters.
Chapter 2

Convex Recoloring

2.1 Introduction

Let $G = (V, E)$ be a graph and let $\chi : V \to C$ be a coloring function, assigning each vertex in $V$ a color in $C$. We say that $\chi$ is a convex coloring of $G$ if, for every color $c \in C$, the vertices with color $c$ induce a connected sub-graph of $G$. Figure 2.1 shows an example of convex and non-convex colorings. In the Convex Recoloring problem (abbreviated CR), we are given a colored graph $G_\chi$, and we wish to find a recoloring of a minimum number of vertices of $G$, such that the resulting coloring is convex. That is, the goal is to find a convex coloring $\chi'$, that minimizes the size of the set $\{v : \chi(v) \neq \chi'(v)\}$. The $t$-Convex Recoloring problem ($t$-CR) is the special case, in which the given coloring assigns the same color to at most $t$ vertices in $G$. Figure 2.2 depicts an input and a possible output for the 2-Convex Recoloring problem.

Related work.

The Convex Recoloring problem (CR) in trees was introduced by Moran and Snir [91] and was motivated by its relation with the concept of perfect phylogeny. They proved that

\footnote{W.l.o.g. we assume that img($\chi$) = $C$.}

Figure 2.1: Example of convex and non-convex colorings of a graph.

(a) Convex

(b) Non-Convex
the problem is NP-hard [62], even when the given graph is a path. Later, Kanj et al. [62] showed that 2-CR is also NP-hard on paths. Applications of CR in general graphs, such as multicast communication, were described by Kammer and Tholey [61]. Many variants of the problem have been intensively investigated. The differences between one variant to another can be related to

- The structure of the given graph $G$. The given graph can be a path, a tree, a bounded treewidth graph, a general graph, and others.

- Constraints on the coloring function $\chi$. In a $t$-coloring at most $t$ colors are used to color a graph, while in a $t$-CR instance at most $t$ vertices are colored using the same color.

- Type of weight function. In the weighted case, each vertex is associated with a weight, and the weight of a recoloring is the total weight of recolored vertices. In the unweighted case the weight of the solution is the number of recolored vertices. In a third variant, referred to as block recoloring [61], a cost is incurred for a color $c$ if at least one vertex of color $c$ was recolored.

Since CR was shown to be NP-hard, it was natural to try to design both approximation algorithms and parameterized algorithms.

Moran and Snir [90] presented a 2-approximation algorithm for CR in paths and a 3-approximation algorithm for CR in trees. Both algorithms work for the problem with weights. Bar-Yehuda, Feldman and Rawitz [17] improved the latter by providing a $(2 + \varepsilon)$-approximation algorithm for CR in trees. This result was later extended to bounded treewidth graphs by Kammer and Tholey [61]. Recently, Lima and Wakabayashi [86] gave a $\frac{3}{2}$-approximation algorithm for unweighted 2-CR in paths.

On the negative side, Kammer and Tholey [61] proved that, if vertex weights are either 0 or 1, then 2-CR has no polynomial time approximation algorithm with a ratio of size $(1 - o(1)) \ln \ln n$ unless $\text{NP} \subseteq \text{DTIME}(n^{O(\log \log n)})$. In Section 2.2, we show that this variant of the problem can not be approximated at all. Campêlo et al. [23] showed that, for $t \geq 2$, CR is NP-hard on $t$-colored grids. They also proved that there is no polynomial time

![Figure 2.2: The 2-Convex Recoloring problem.](image)
approximation algorithm within a factor of $c \ln n$ for some constant $c > 0$, unless $P = NP$, for unweighted CR in bipartite graphs with 2-colorings.

Moran and Snir [91] presented an algorithm for CR whose running time is $O(n^4 k (\frac{k}{\log k})^k)$, where $k$ is the number of recolorings in an optimal solution. Razgon [99] gave a $2^{\mathcal{O}(k)} \text{poly}(n)$ time algorithm for CR in trees. Ponta et al. [98] designed several algorithms for different variants of CR in trees. For the unweighted case, they gave a $O(3^k \cdot b \cdot n)$ time algorithm, where $b$ is the number of colors that do not induce a connected subtree, and it is bounded from above by $2k$. Bar-Yehuda, Feldman and Rawitz [17] provided an algorithm with an upper bound of $O(n^2 + n \cdot k^2k)$ on the running time, but it is not hard to verify that this bound can be improved to $O(n \cdot k^2k)$. Bodlaender et al. [18] showed that CR admits a kernel of size $O(k^2)$ in trees. Bachoore and Bodlaender [14] presented an algorithm for leaf-colored trees with a running time of $O(4^k \cdot n)$. Campêlo et al. [23] proved that, for $t \geq 2$, CR is $W[2]$-hard in $t$-colored graphs.

Our results.

Section 2.2 contains our hardness result for the weighted version of 2-CR. We show that even the 2-CR feasibility question is NP-hard in the case of binary weights. In Section 2.3 we provide an alternative definition of 2-CR in terms of maximal independent set of paths. We first show that we can focus on a specific type of recoloring, called a path recoloring, in which each color induces a path. Then we show that finding a path recoloring can be translated into finding an independent set of paths. In Section 2.4 we present a greedy algorithm for (unweighted) 2-CR in general graphs that is based on iteratively adding a shortest path to the current independent set of paths. We provide a tight analysis for the algorithm and show that its approximation ratio is $\frac{3}{7}$. This is the first time a constant-ratio approximation algorithm is given for a variant of CR in general graphs. We also show that when $G$ is a path, the same algorithm yields a $\frac{4}{3}$-approximation, improving the previous best known approximation ratio by Lima and Wakabayashi [86]. In Section 2.5 we use the above mentioned characterization of 2-CR to show that a problem kernel of size $4k$ can be obtained in linear time, where $k$ is the cost of an optimal solution. This leads to a $O(|E|) + 2^{\mathcal{O}(k \log k)}$ time algorithm for 2-CR.

2.2 Hardness Result

In this section we prove that the weighted version of 2-CR cannot be approximated within any multiplicative ratio, unless $P=NP$. We do so using a simple reduction from the DISJOINT CONNECTING PATHS problem.

In the DISJOINT CONNECTING PATHS problem the input consists of an undirected graph $G = (V, E)$ and a set of pairs $\{(s_1, t_1), \ldots, (s_k, t_k)\}$, and the question is whether there are $k$ vertex-disjoint paths that connect $s_i$ to $t_i$. This problem is known to be NP-hard [64].

**Theorem 1.** The weighted version of 2-CR cannot be approximated within any multiplicative ratio, unless $P=NP$.

**Proof.** Let $G$ and $\{(s_1, t_1), \ldots, (s_k, t_k)\}$ be an instance of DISJOINT CONNECTING PATHS. We constructed a weighted 2-CR instance as follows. First, we use the same graph $G$. Let $\chi(v) = c_v$ (a unique color assigned only to vertex $v$), if $v \in \{s_1, \ldots, s_k\} \cup \{t_1, \ldots, t_k\}$, and let $\chi(s_i) = \chi(t_i) = c_i$, for every $i$, where $c_1, \ldots, c_k$ are distinct colors. Define a weight function
2.3 Properties of Optimal Recolorings

In this section we introduce a special type of convex recoloring, called path-recoloring. Path-recolorings are more constrained than general recolorings and thus are simpler to understand and analyze. Nevertheless, we show that, for the 2-CR problem, there is always an optimal convex recoloring that is a path-recoloring. Based on the above, we give an alternative definition to 2-CR in terms of independent set of paths. From now on, we only consider the 2-CR problem, in particular, whenever we mention a colored graph, we refer to a 2-CR instance.

Given a colored graph $G_x$, if two vertices in the colored graph have the same color, we call them a pair; if they are connected with an edge, then they are a connected pair, otherwise they are a disconnected pair. Any vertex with a unique color is a singleton, we call $c$ a singleton color. We denote by $G_x[c]$ the subgraph induced by the set of vertices $\{v : \chi(v) = c\}$. Figure 2.3 depicts these concepts.

### 2.3.1 Path-Recoloring

As a first step we show that it may be assumed that all colors retain at least one representative. In other words, we show that there is always an optimal convex recoloring $\chi'$ that does not recolor singletons and recolors at most one vertex of every pair.

Given a colored graph $G_x$ and a recoloring $\chi'$, a vertex $v$ retains its color if $\chi'(v) = \chi(v)$. We say that $\chi'$ retains a pair $p$, if both vertices of $p$ retain their color. The recoloring $\chi'$ retains a color $c \in C$, if there exists a vertex $v \in G$ such that $\chi'(v) = \chi(v) = c$. If a recoloring retains all the colors of a graph, we refer to it as a retains-all recoloring (see Figure 2.4). Observe that a retains-all recoloring does not recolor singletons.

**Lemma 1.** For every colored graph $G_x$, there exists a retains-all optimal convex recoloring.

**Proof.** Consider an optimal convex recoloring $\chi'$ that retains a maximum number of colors over all optimal, convex recolorings of $G$. Assume for contradiction that $\chi'$ does not retain

---

Figure 2.3: In this colored graph, vertex 5 is a singleton, vertices 1 and 2 are a connected pair, and vertices 3 and 4 are a disconnected pair.

The function $w$ is defined as follows: $w(v) = 1$, if $v \in \{s_1, \ldots, s_k\} \cup \{t_1, \ldots, t_k\}$, and $w(v) = 0$, otherwise. It is not hard to verify that $k$ disjoint paths exist in $G$ if and only if the minimum convex recoloring costs zero. Hence, it is NP-hard to decide whether a zero cost solution exists.

---

Note that this definition is different from the one given in [62].
For every colored graph $G$, consider an optimal retains-all convex recoloring $\chi'$. Without loss of generality we assume that $c$ is not used by $\chi'$ (otherwise, we can recolor each vertex in $G_{\chi'}[c]$ using a new unique color. We define a recoloring $\chi''$ by considering $G_{\chi'}[c]$. First, define $\chi''(v) = c$. Next, if $G_{\chi'}[c']$ contains a vertex $u$ such that $\chi(u) = c'$, then define $\chi''(u) = c'$. Each vertex of the remaining vertices in $G_{\chi'}[c']$ is colored by $\chi''$ using a unique new color. Observe that if there exists a second vertex $u' \neq u$ such that $\chi(u') = \chi(u') = c'$, then $\chi''$ recolors it. Finally, $\chi''(x) = \chi'(x)$ for any vertex $x$ not in $G_{\chi'}[c']$.

$\chi''$ is convex since every vertex in $G_{\chi'}[c']$ is colored by a unique color. $\chi''$ recolors at most as many vertices as $\chi'$ since it may recolor $u$'s mate (if it exists), but it avoids the recoloring of $v$. Finally, $\chi''$ retains more colors than $\chi'$ since it retains $c$. Thus, we obtained an optimal recoloring that retains more colors than $\chi'$. A contradiction.

Next we show that we need not recolor connected pairs.

**Lemma 2.** For every colored graph $G_\chi$ there exists a retains-all, optimal, convex recoloring that does not recolor any connected pair.

**Proof.** Consider an optimal retains-all convex recoloring $\chi'$ that retains the maximum number of connected pairs, over all optimal, retains-all recolorings of $G$. Assume for contradiction that $\chi'$ recolors one of the vertices of a connected pair $\{u, v\}$, that is, without loss of generality, $\chi(u) = \chi(v) = c$ and $\chi'(v) = c' \neq c$. We can use a similar argument to the one used in the proof of Lemma 1. The difference is in the definition of $\chi''$. We let $\chi''(v) = c$. Next, if $G_{\chi'}[c']$ contains a vertex $w \neq v$ such that $\chi(w) = c'$, then define $\chi''(w) = c'$, also, if $G_{\chi'}[c']$ contains a connected pair $u_1, u_2$ such that $\chi(u_1) = \chi(u_2) = \chi'(u_1) = \chi'(u_2) = c'$, then we define $\chi''(u_1) = \chi''(u_2) = c'$. Each vertex of the remaining vertices in $G_{\chi'}[c']$ is colored by $\chi''$ using a unique new color. Observe that $\chi''$ is convex, since apart from the possible connected pair $u_1$ and $u_2$ which are colored by $c'$, every vertex in $G_{\chi'}[c']$ is colored by a unique color. Observe also that $\chi''$ is a retains-all recoloring that recolors at most as many vertices as $\chi'$, and retains more connected pairs than $\chi'$. A contradiction.

We are now ready to define path-recolorings. Given a colored graph $G_\chi$ and a convex recoloring $\chi'$, we say that $\chi'$ path-recolors $G$ with respect to $c \in C$ if there is a Hamiltonian path.
path $\vec{B}$ in $G_{\chi'}[c]$: $u, \ldots, v$ such that $\chi(u) = \chi(v) = c$. A special case of this definition is when $G_{\chi'}[c]$ is a single vertex $v$ and $\chi(v) = c$. We say that $\chi'$ is a path-recoloring if:

1. $\chi'$ does not recolor any connected pair, and
2. $\chi'$ path-recolors $G$ with respect to every $c \in C$.

Clearly, every path-recoloring also retains all colors.

**Lemma 3.** For every colored graph $G_{\chi}$ there exists an optimal recoloring that is a path recoloring.

**Proof.** Let $\chi'$ be an optimal, retains-all recoloring that does not recolor any connected pair (whose existence was shown in Lemma 2) that path-recolors $G$ with respect to a subset of $C$ of a maximum possible size. Assume for contradiction that $\chi'$ is not a path-recoloring. Hence there is a color $c$ such that $\chi'$ does not path-recolor $G$ with $c$. Consider $G_{\chi'}[c]$, and assume for now that there are two vertices $u$ and $v$ in $G_{\chi'}[c]$ such that $\chi(u) = \chi(v) = c$. Fix a simple path from $u$ to $v$ in $G_{\chi'}[c]$, and let $\chi''$ be identical to $\chi'$ with the following modification: $\chi''$ assigns a unique color for every vertex in $G_{\chi'}[c]$ that is not on the simple path from $u$ to $v$. Clearly, $\chi''$ is an optimal recoloring that path-recolors $G$ with respect to more colors than $\chi'$. A contradiction.

Now, if there is at most one vertex $v$ in $G_{\chi'}[c]$ such that $\chi(v) = c$ then consider a recoloring $\chi''$ that is identical to $\chi'$ except it assigns a unique color to every vertex in $G_{\chi'}[c]$ that is not $v$. This time, again, we found an optimal recoloring that path-recolors $G$ with more colors than $\chi'$. A contradiction.

Henceforth, whenever we refer to a recoloring, we assume that it is a path-recoloring. In particular, we assume that for every disconnected pair in $G$ with color $c$, a recoloring either: (i) colors exactly a path between the disconnected pair with color $c$, or (ii) changes the color of exactly one of its vertices.

### 2.3.2 Path Independence

When a recoloring $\chi'$ colors a path between a disconnected pair (using the pair’s color), we refer to the path as a colored path. Let $D$ be the set of all disconnected pairs in $G$, and denote by $I$ the set of colored paths in $G_{\chi'}$. Then the following lemma holds:

**Lemma 4.** Given a colored graph $G_{\chi}$, a path-recoloring $\chi'$ recolors exactly $|D| - |I|$ vertices.

**Proof.** By definition, $\chi'$ may only recolor vertices from the disconnected pairs. Also, it does not recolor any of the vertices that form the endpoints of paths in $I$. $\chi'$ must recolor exactly one vertex of every other disconnected pair: if it recolors both vertices then the color of this pair is not retained, and if it recolors none of them then convexity does not hold.

Given a colored graph $G_{\chi}$ and a path $p$, let $V(p)$ be the set of vertices on the path and let $\chi(p)$ be the set of colors assigned to vertices on this path, i.e. $\chi(p) = \{ \chi(v) : v \in V(p) \}$. Given two paths $p_1$ and $p_2$ in $G_{\chi}$:

- $p_1$ and $p_2$ are in direct conflict if $V(p_1) \cap V(p_2) \neq \emptyset$.

\[ \text{Note that we are not concerned with the decision problem whether a given recoloring is a path-recoloring or not. Later in the thesis we will introduce a simple algorithm that computes a path-recoloring.} \]
Figure 2.5: In this colored graph, the paths \((4, 6, 3)\) and \((1, 5, 2)\) are colorable, while the path \((6, 7, 5)\) is not. Path \((4, 6, 3)\) is in indirect conflict with path \((1, 5, 2)\) and in direct conflict with path \((6, 7, 5)\).

- \(p_1\) and \(p_2\) are in indirect conflict if \(\chi(p_1) \cap \chi(p_2) \neq \emptyset\),
- \(p_1\) and \(p_2\) are in conflict if they are either in a direct or an indirect conflict (observe that direct conflict implies an indirect conflict). If two paths are not in conflict, then they are independent. Given a set of paths \(I\), we say that this set is independent if it is pairwise independent, that is, if every two paths \(p_1, p_2 \in I\) are independent. A path \(u, \ldots, v\) in \(G\) is called colorable if \(u\) and \(v\) form a disconnected pair and the path does not contain singletons nor vertices of connected pairs. Figure 2.5 depicts these concepts.

In the next lemma we show that the set of colored paths that is induced by a path-recoloring is an independent set of colorable paths.

**Lemma 5.** Let \(G_\chi\) be a colored graph. Also, let \(\chi'\) be a path-recoloring, and let \(I\) denote the set of colored paths (w.r.t. \(\chi'\)). Then \(I\) is an independent set of colorable paths with respect to \(\chi\).

**Proof.** Obviously, two colored paths in \(I\) cannot be in direct conflict. Assume for contradiction that there are two paths \(p_1, p_2 \in I\) that are in indirect conflict, that is, there is a color \(c \in \chi(p_1) \cap \chi(p_2)\). It follows that \(\chi'\) recolors two vertices of the same color, and we get a contradiction since \(\chi'\) does not retain all colors. Finally, it follows that all colored paths, with respect to \(\chi'\), are colorable with respect to \(\chi\), or otherwise \(\chi'\) must recolor a singleton or a connected pair.

We say that a set of paths, \(I\), covers a pair if at least one of the vertices of the pair belongs to one of the paths in \(I\).

**Lemma 6.** For any independent set of colorable paths \(I\) in \(G_\chi\), there exists a path-recoloring \(\chi'\) of \(G\), where \(I\) is the set of colored paths.

**Proof.** Consider a recoloring \(\chi'\) that colors every colorable path in \(I\) using the color of its endpoints, assigns a unique new color to one (arbitrary) of the endpoints of every disconnected pair that is not covered by \(I\) and keeps the colors of the other vertices. We first show that \(\chi'\) is a path recoloring. Since \(I\) is independent, no two paths in \(I\) contain the same color. Moreover, every path in \(I\) is colorable, thus, by definition, it does not contain singletons. It follows that \(\chi'\) retains all colors. Also, every path in \(I\) does not contain any vertices of connected pairs, thus \(I\) does not recolor vertices of connected pairs. Finally, by the construction of \(\chi'\), it is not hard to verify that for every color \(c \in C\), the subgraph \(G_{\chi'}[c]\) contains a Hamiltonian path or it is a single vertex. Further observe that for every color \(c \in \text{Image}(\chi')\), \(G_{\chi'}[c]\) is either a simple path or a single vertex, and it follows that \(\chi'\) is a
path-recoloring. Finally, by the construction of \( \chi' \), it is straightforward to verify that \( I \) is the set of colored paths induced by \( \chi' \).

The following is obtained due to Lemmas 4, 5 and 6.

**Theorem 2.** Given a colored graph \( G_\chi \), the cost of an optimal (path-)recoloring is \( |D| - s \) if and only if the size of the maximum independent set of colorable paths is \( s \).

Theorem 2 suggests an alternative definition to 2-CR: given a colored graph \( G_\chi \), find a maximum independent set of colorable paths in \( G_\chi \).

### 2.4 Greedy Algorithm

In this section we describe a natural greedy algorithm to construct a maximal independent set of colorable paths, we discuss how this algorithm can be implemented, and define the corresponding path-recoloring it computes. We show that the approximation ratio of the greedy algorithm is \( \frac{3}{2} \). For the special case where the input graph is a path, we show that the greedy algorithm achieves a \( \frac{5}{4} \) approximation ratio.

#### 2.4.1 The Algorithm

While we are not attempting to achieve an approximation to the size of the maximum independent set of colorable paths, the alternative definition given at the end of the previous section leads us to a natural greedy algorithm: choose the shortest colorable path that is not in conflict with colorable paths already been chosen and add it to an independent set of colorable paths. A formal description of this algorithm is given in Algorithm 1.

**Algorithm 1:** Greedy algorithm for 2-CR.

1. \( I \leftarrow \emptyset \)
2. while there is a colorable path, not in conflict with \( I \) do
   3. add to \( I \) a shortest colorable path, not in conflict with \( I \)
4. end
5. return the path-recoloring corresponding to \( I \)

We now describe how a shortest colorable path can be found. To do that, at the initialization of the algorithm, all singletons and connected pairs should be removed from the graph. Every colorable path that is added to \( I \) should be also removed from the graph. In addition, after each path removal, one should also remove all vertices with a unique color in the remaining graph. On the remaining graph, a shortest path, among those vertices of the same color, is guaranteed to be colorable and independent of \( I \). In particular, it cannot contain two vertices that are colored by the same color, since such a path is not a shortest path.

#### 2.4.2 Analysis for General Graphs

We now analyze the greedy algorithm and prove that it recolors at most \( \frac{3}{2} \cdot k \) vertices, where \( k \) is the minimum number of vertices that must be recolored to achieve a convex coloring. We also show that the analysis is tight even in trees.
Recall that $D$ is the set of disconnected pairs in $G$, $I \subseteq D$ is the set of independent paths colored by the greedy algorithm, and let $I^* \subseteq D$ be the set of independent paths colored by an arbitrary, but fixed, optimal path-recoloring. Define $\alpha := \frac{|I^*|}{|I|}$, then by Lemma 4 the ratio between the number of vertices recolored by the greedy algorithm and the number of vertices recolored by an optimal path-recoloring is:

$$r := \frac{|D| - |I|}{|D| - |I^*|} = \frac{|D| - |I|}{|D| - \alpha |I|}$$

Recall that due to Lemma 2 we may assume that an optimal solution does not recolor singletons or disconnected pairs. We now analyze the relationship between $|I|$, $|D|$ and $\alpha$.

Let $\ell_p$ be the number of vertices on a path $p$ (including its endpoints), and observe that:

**Lemma 7.** If $p' \in I^* \setminus I$, then there is a path $p \in I$ that is in conflict with $p'$, and $\ell_p \leq \ell_{p'}$.

*Proof.* Consider the running of the greedy algorithm at the most recent point, where $I$ contains no path with a distance greater than $\ell_{p'}$. If, at this point, there is no path in $I$ that is in conflict with $p'$ then nothing prevents the greedy algorithm from adding $p'$ to $I$. \qed

Next, we assign each optimal path to a greedy path. Given a path $p' \in I^*$, the conflict source of $p'$ is $p'$ itself if $p' \in I$, otherwise it is an arbitrary shortest path in $I$ that is in conflict with $p'$. Notice that this is a many-to-one assignment, that is, several optimal paths can be assigned to a single greedy path. For a path $p \in I$, the set of paths in $I^*$ such that $p$ is their conflict source is denoted as $N(p)$. The members of $N(p)$ are called the neighbours of $p$. Figure 2.6 depicts the mapping between paths in $I^*$ and their conflict source in $I$.

Due to Lemma 7 we have that:

**Observation 1.** For every path $p \in I$, if $p' \in N(p)$, then $\ell_p \leq \ell_{p'}$.

Denote $d_p := |N(p)|$ and refer to $d_p$ as the degree of $p$. Our next goal is to find an upper bound to $d_p$, for $p \in I$.

**Lemma 8.** For every path $p \in I$, $d_p \leq \ell_p - 1$.

*Proof.* We show that the number of paths in $I^*$ that are in conflict with $p$ is at most $\ell_p - 1$. Associate a vertex in $p$ to every path that is in conflict with $p$. For a path that is in direct conflict with $p$ associate a common vertex of these paths, for a path that is in indirect conflict due to color $c$, associate the vertex that has color $c$ in $p$. Recall that no two paths in $I^*$ are in conflict, and observe that associating the same vertex to more than one path implies the existence of such conflict. Finally, since the end points of $p$ have the same color, at most one of them can be associated with a path in $I^*$, or otherwise, $I^*$ is not independent. \qed

In what follows we obtain two lower bounds on $|D|$, which translate into two upper bounds on the approximation ratio $r$.

**Lemma 9.** $|D| \geq 2|I^*|$.

*Proof.* Every path $p \in I^*$ has $\ell_p \geq 3$, since its end points form a disconnected pair. Also, the internal vertices must be part of another disconnected pair. Thus, we can observe the existence of at least two disconnected pairs for every path in $I^*$. We do not count the same disconnected pair twice, or otherwise, $I^*$ is not independent. \qed
Figure 2.6: A comparison between $I$ (the greedy solution) and $I^*$ (an optimal solution). Each path in $I^*$ is mapped to its conflict source in $I$. 
Observation 2. $\sum_{p \in I} d_p = |I^*| = \alpha \cdot |I|$

Proof. By definition every path $p' \in I^*$ has one, and only one, conflict source in $I$, thus, there exists exactly one path $p \in I$ such that $p' \in N(p)$. \hfill $\square$

Lemma 10. $\frac{\sum_{p \in I} d_p^2}{|I|} \geq \alpha^2$.

Proof. Due to Observation 2, we have that the average degree of paths in $I$ is $\alpha$, i.e. $\frac{\sum_{p \in I} d_p}{|I|} = \alpha$. The lemma follows from the Jensen’s inequality. \hfill $\square$

Lemma 11. $|D| \geq \alpha^2 |I|$

Proof. Let $v$ be a vertex in a path $p' \in I^*$. Then $v$ must be a part of a disconnected pair, otherwise $v$ is a singleton or part of a connected pair and thus the considered recoloring is not a path-recoloring. Observe, also, that aside from the two endpoints of each path, no two vertices of the same disconnected pair can belongs to paths in $I^*$ (or otherwise $I^*$ is not independent). Thus, we can count $\ell_{p'} - 1$ disconnected pairs for every path $p'$ in $I^*$. Hence

$$|D| \geq \sum_{p' \in I^*} (\ell_{p'} - 1) = \sum_{p' \in I^*} \ell_{p'} - |I^*|$$

Consider a path $p \in I$. Recall from Observation 1 that if $p' \in N(p)$ then $\ell_{p'} \geq \ell_p$. Hence, $\sum_{p' \in N(p)} \ell_{p'} \geq d_p \cdot \ell_p$, and it follows that

$$\sum_{p' \in I^*} \ell_{p'} \geq \sum_{p \in I} d_p \cdot \ell_p \geq \sum_{p \in I} d_p(d_p + 1) = \sum_{p \in I} d_p^2 + |I^*|$$

where the second inequality is due to Lemma 8 and the equality is due to Observation 2. Hence, by Lemma 10 we have that $|D| \geq \sum_{p \in I} d_p^2 \geq \alpha^2 |I|$. \hfill $\square$

We can now obtain two upper bounds on the approximation ratio $r$ as a function of $\alpha$.

Theorem 3. The greedy algorithm is a $3 \cdot \alpha^{-1}$-approximation algorithm for 2-CR.

Proof. Using Lemma 11 and the fact that $\alpha \geq 1$ we get that

$$r = \frac{|D| - |I|}{|D| - \alpha \cdot |I|} \leq \frac{\alpha^2 \cdot |I| - |I|}{\alpha^2 \cdot |I| - \alpha \cdot |I|} = \frac{\alpha^2 - 1}{\alpha^2 - \alpha} = \frac{\alpha + 1}{\alpha}$$

and from Lemma 9 we get that

$$r = \frac{|D| - |I|}{|D| - \alpha \cdot |I|} \leq \frac{2\alpha \cdot |I| - |I|}{2\alpha \cdot |I| - \alpha \cdot |I|} = \frac{2\alpha - 1}{\alpha}$$

Figure 2.7 depicts the two upper bounds on $r$. Putting the two bounds together, it follows that

$$r \leq \min \left\{ \frac{\alpha + 1}{\alpha}, \frac{2\alpha - 1}{\alpha} \right\} \leq \frac{3}{2}$$

as required. \hfill $\square$

We show that our analysis is tight even for colored trees, using the instance depicted in Figure 2.8, which consists of a colored graph where the greedy algorithm might recolor $\frac{3}{2}$ times more vertices than the optimal convex recoloring. We note that one can simply duplicate the instance using new colors for each copy in order to construct an arbitrary large graph with the same approximation ratio.
Figure 2.7: An upper bound on the approximation ratio as a function of $\alpha$. The dashed (blue) and the dotted (orange) plots are the two bounds for the general case. The solid (green) plot is the bound for 2-cr on paths.

Figure 2.8: Greedy might choose to color the path $(1, 3, 2)$, then it must recolor one of the vertices $(5, 6)$ and one of the vertices $(7, 8)$, a total of three recolored vertices, while an optimal recoloring can color two paths: $(5, 1, 6)$ and $(7, 3, 8)$, a total of two recolored vertices.
2.4.3 Analysis for Paths

For the special case when the input graph is a path, we show that the greedy algorithm achieves a $\frac{5}{4}$ approximation ratio. We show that the analysis is tight for this special case as well.

We start our analysis by observing that, in the case of a path, we can replace Lemma 8 with the following lemma:

**Lemma 12.** For every path $p \in I$, $d_p \leq \ell_p - 2$.

**Proof.** We show that the number of paths in $I^*$ that are in conflict with $p$ is at most $\ell_p - 2$. This is true, because if $G$ is a path, then for every path that is in conflict with $p$, we can now associate a (unique) vertex that is not one of the endpoints of $p$. If the conflict is direct, then the two paths must overlap, thus associate the other path’s endpoint. If the conflict is indirect, then this must be due to some color other than the one on the endpoints of $p$, or else this is a direct conflict.

Using the above lemma we can strengthen Lemma 11.

**Lemma 13.** $|D| \geq (\alpha^2 + \alpha)|I|$.

**Proof.** The proof is similar to the proof of Lemma 11, where the main difference is that

$$\sum_{p' \in I^*} \ell_{p'} \geq \sum_{p \in I} d_p \cdot \ell_p \geq \sum_{p \in I}(d_p^2 + 2d_p) = \sum_{p \in I} d_p^2 + 2|I^*|$$

which means that

$$|D| \geq \sum_{p' \in I^*} \ell_{p'} - |I^*| \geq \sum_{p \in I} d_p^2 + |I| \geq \sum_{p \in I} d_p^2 + \alpha|I| \geq (\alpha^2 + \alpha)|I|$$

and the lemma follows.

In this case only one upper bound suffices.

**Theorem 4.** Greedy is a $\frac{5}{4}$-approximation algorithm for 2-CR on paths.

**Proof.** Using Lemma 13 we get that

$$r = \frac{|D| - |I|}{|D| - \alpha \cdot |I|} \leq \frac{(\alpha^2 + \alpha)|I| - |I|}{(\alpha^2 + \alpha)|I| - \alpha|I|} = \frac{\alpha^2 + \alpha - 1}{\alpha^2} = \frac{5}{4} - \frac{\alpha^2}{\alpha^2} \leq \frac{5}{4}$$

as required.

Figure 2.7 compares this last bound with the bounds of the general case. To see that our analysis is tight, consider the instance depicted in Figure 2.9. On this colored path, the greedy algorithm might recolor $\frac{5}{4}$ times more vertices than the optimal recoloring.
Figure 2.9: The greedy algorithm might choose to color the path 2...5, then it must recolor one of the vertices \{1, 10\} (the path 1...10 contains the path 2...5), one of \{6, 12\} (the paths 2...5 and 6...12 are in indirect conflict), and one of \{8, 11\} (the paths 2...5 and 8...11 are in indirect conflict), a total of five recolored vertices, while an optimal recoloring can recolor the paths 4...7 and 8...11, a total of four recolored vertices.

### 2.5 Parameterized Complexity

Let \( k \) be the minimum number of vertices that has to be recolored. In this section, we describe an exact algorithm for 2-CR with a running time of \( O(|E|) + 2^{O(k \log k)} \).

**Theorem 5.** There is a 4\(k\) kernel for 2-CR that can be computed in linear time, where the parameter \( k \) is the size of an optimal solution.

**Proof.** Recall from Lemma 4 that \( k = |D| - |I^*| \). From Lemma 9 we know that \( |D| \geq 2|I^*| \) and we can conclude that \( |D| \leq 2k \). That is, the number of disconnected pairs in any colored graph is at most twice the minimum number of vertices needed to be recolored by any convex recoloring. Finally, Lemma 3 states that an optimal solution that recolors only disconnected pairs exists. Thus, given a colored graph, we can reduce, in linear time, the number of vertices in the graph to no more than \( 4k \) by removing all singletons and connected pairs. \( \square \)

We use kernelization to obtain a parameterized algorithm for 2-CR.

**Theorem 6.** There is an \( O(|E|) + 2^{O(k \log k)} \) time algorithm for 2-CR, where the parameter \( k \) is the size of an optimal solution.

**Proof.** First reduce the size of the graph to no more than \( 4k \) vertices, as described in the proof of Theorem 5. This takes \( O(|E|) \). A brute force search algorithm that tests every possible recoloring can now be used to find an optimal one. Such algorithm needs to consider \( 2^k \) colors for every vertex of the \( 4k \) vertices, thus at most \( (2^k)^{4k} \) coloring exists. Each proposed recoloring can be tested in linear time, so the overall running time of the algorithm is \( O(k^2 \cdot (2^k)^{4k}) = 2^{O(k \log k)} \). \( \square \)
Chapter 3

Service Chain Placement in SDNs

3.1 Introduction

Computer communication networks are in constant need of expansion to cope with the ever growing traffic. As networks grow, management and maintenance become more and more complicated. Current developments that aim to improve the utilization of network resources include the detachment of network applications from network infrastructure and the transition from network planning to network programming.

One aspect of network programming is to manage resources from a central point of view, namely to make decisions based on availability, network status, required quality of service, and the identity of the client. Hence, a focal issue is a central agent that is able to receive reports from network components and client requests, and as a result can alter the allocation of resources in the networks. This approach is called Software Defined Networking (SDN), where there is a separation between routing and management (control plane) and the underlying routers and switches that forward traffic (data plane) (see Kreutz et al. [81]).

A complementary approach is Network Function Virtualization (NFV) [36]. Instead of relying on special purpose machines, network applications become virtual network functions (VNF) that are executed on generic machines and can be placed in various locations in the network. Virtualization increases the flexibility of resource allocation and thus the utilization of the network resources. Internet Service Providers (ISPs) that provide services to clients benefit from NFV, since it helps to better utilize their physical network. In addition, when network services are virtualized, an ISP may support service chaining [19], namely a compound service that consists of a sequence of VNFs. Furthermore, one may offer a compound service that may be obtained using one of several sequences of VNFs.

3.1.1 Related Work

In this paper we consider an SDN network that employs NFV [37]. Such networks attracted a lot of attention in the networking community, especially from a systems point of view (see, e.g., [50, 56]). Several papers also considered the algorithmic aspects of such networks but mainly presented heuristics (see, e.g., [105]). For more details see [40] and references therein.
Cohen et al. [28] considered VNF placement. In their model the input is an undirected graph with a metric distance function on the edges. Clients are located in various nodes, and each client is interested in a subset of VNFs. For each node $v$ and VNF $\alpha$, there is a setup cost associated with placing a copy of $\alpha$ at $v$ (multiple copies are allowed), and there is a load that is induced on $v$ for placing a copy of $\alpha$. Furthermore, each node has a limited capacity, and each copy of a VNF can handle a limited amount of clients. A solution is the assignment of each client to a subset of nodes, each corresponding to one of its required VNFs. The cost of a solution is the total setup costs plus the sum of distances between the clients and the node from which they get service. Cohen et al. [28] gave bi-criteria approximation algorithms for various versions of the problem, namely algorithms that compute constant factor approximations that violate the load constraints by a constant factor. It is important to note that in this problem routing is not considered and it is assumed that VNF subsets can be executed in any order.

Lukovszki and Schmid [89] studied the problem of admission control and placement of service chains, where each chain is associated with a source-destination pair and is supposed to be routed via an ordered sequence of $\ell$ VNFs. The VNFs may have multiple instantiations, but each node has a limited capacity that bounds the number of requests it may service. They presented an $O(\log \ell)$-competitive algorithm for the problem of maximizing the number of serviced chains, assuming that capacities are $\Omega(\log \ell)$. It is also shown that this ratio is asymptotically optimal even for randomized online algorithms. APX-hardness results for the offline version of the problem were also presented. Sahhaf et al. [102] also studied placement of network service chains. They presented an algorithm which minimizes the cost of the placement and is based on integer linear programming. They also present a heuristic in order to cope with the non-practical running time of the first algorithm. Rost and Schmid [100] considered variant of this problem, where each node can host a subset of the VNFs. In addition, each VNF has a demand and each VNF-node pair has a capacity, and in a feasible solution the total demand for each pair is bounded by the capacity. They considered two goals: maximum profit and minimum resource cost and gave bicriteria approximation algorithms which are based on LP-rounding for several special cases. Rost and Schmid [101] considered a more general version of the problem in which requests are graphs that should be placed within the physical graph. They provided hardness of approximation results for this problem.

Even, Medina, and Patt-Shamir [38] (see, also [39]) studied online path computation and VNF placement. They considered compound requests that arrive in an online manner. Each request is a flow with a specification of routing and VNF requirements which is represented by a directed acyclic graph (DAG) whose vertices are VNFs. Each VNF can be performed by a specified subset of servers in the system. Upon arrival of a request, the algorithm either rejects the request or accepts it with a specific routing and VNF assignment, under given server capacity constraints. Each request has a benefit that is gained if it is accepted, and the goal is to maximize the total benefit gained by accepted requests. Even, Medina, and Patt-Shamir [38] proposed an algorithm that copes with requests with unknown duration without preemption by using a third response, which is refer to as “stand by”, whose competitive ratio is $O(\log(kn b_{\text{max}}))$, where $n$ is the number of nodes, $b_{\text{max}}$ is an upper bound on a request benefit per time unit, and $k$ is the maximum number of VNFs of any service chain. This performance guarantee holds for the case where the processing of any request in any possible service chain is never more than $O(1/(k \log(nk)))$ fraction of the capacity of any network component. Even, Rost, and Schmid [40] presented a randomized approximation algorithm for the same problem that computes $(1-\varepsilon)$-approximate placements if the demand of any
processing request in any possible service chain is never more than an $O(\varepsilon^2/\log n)$ fraction of the capacity of any network component.

Foerster, Parham and Schmid [47] considered the Bidirectional Waypoint Routing problem, where the input consists of a graph $G = (V, E)$ with edge weights and capacities, a source $s$, a destination $t$, and a subset of the vertices $W$. A feasible solution is a route from $s$ to $t$ that visits the vertices in $W$. If $W$ is ordered, the route should visit the vertices of $W$ according to the given order. The goal is to find a minimum weight solution. Foerster et al. [47] showed that the unordered problem has no PTAS, assuming $P \neq NP$ and gave a $1.53$-approximation algorithm. They also presented polynomial time algorithms for special cases. As for the ordered variant, they showed that feasible routes can be computed efficiently if $|W| = O(1)$. In addition, they showed that the problem is NP-hard and gave a polynomial time algorithm for cactus graphs with constant link capacities. Amiri et al. [9] considered the unordered case, and they presented a polynomial-time algorithm for graphs of bounded treewidth. Amiri et al. [7] studied the ordered case, and they gave algorithms for special networks. An overview of waypoint routing is given in [8].

3.1.2 Our Results

We study the allocation problem of a compound flow request for a service chain in a software-defined network that supports NFV, where the goal is to find an optimal placement with respect to the current available resources from the point of view of the network (i.e., the ISP). More specifically, we consider the case where the network already contains previous resource allocations. Therefore, we are given a (physical) network that contains servers with limited (residual) processing power and links with limited (residual) bandwidth. A request for service is composed of a source and a destination in the network, an upper bound on the total latency, and a specification of all service chains that are considered valid for this request. As in [38], the specification is represented by a DAG whose vertices are VNFs. The allocation of a service chain consists of routing and VNF placement. That is, each VNF from the sequence is placed in a server along a path, and it is feasible if each server can handle the VNFs that are assigned to it, and if each link on the path can carry the flow that is assigned to it. Moreover each link causes a delay, and the service chain placement should also comply with a global bound on the total latency. Each pair of server and VNF is associated with a cost for placing the network function in the server. This cost measures compatibility of a VNF to a server (e.g., infinite costs means “incompatible”). Given a request, the goal is to find a feasible service chain of minimum total cost or to identify that a valid service chain does not exist.

We show that even feasibility is NP-hard in general networks using a reduction from Hamiltonian Path. We show that the problem is still NP-hard if the network is a DAG even for a very simple network, and even if the specification consists of a single option. Both reductions are from Partition. On the positive side, we present an FPTAS for the case where the network is a DAG which is based on a non-trivial dynamic programming algorithm. Based on our FPTAS, we provide a randomized algorithm for general networks in which there is an optimal placement whose physical path consists of at most $k$ vertices whose degree is larger than 2. For example, this can be assumed if all simple paths from $s$ to $t$ contain at most $k$ such vertices. The algorithm computes a $(1 + \varepsilon)$-approximate placement with high probability using $k! \log n$ invocations of the FPTAS. We also present a (deterministic) parameterized algorithm that computes a $(1 + \varepsilon)$-approximate placement in time $O(k! \cdot \text{poly}(n))$, where the parameter $k$ is the number of vertices in the network whose
We note that there are special cases (e.g., cactus graphs) in which the running time of the algorithm becomes polynomial. Finally, we provide an FPTAS for a fault-tolerant version of the service chain placement problem (without latency) in which two virtual paths are placed in two vertex joint paths.

3.2 Preliminaries

We formally define the problem in this section.

3.2.1 Model

An instance of the Service Chain Placement (SCP) problem is composed of three components, a physical network, a virtual specification, and placement costs:

**Physical network:** The physical network is a graph $G = (V, E)$. Each node $v \in V$ has a non-negative processing capacity $p(v)$, and each directed edge $e \in E$ has a non-negative bandwidth capacity $b(e)$.

**Virtual specification:** The description of a request for a service chain consists of a physical source $s \in V$, a physical destination $t \in V$, and a directed acyclic graph (DAG) $G = (V, E)$. Without loss of generality, we assume that $p(s) = p(t) = 0$. The DAG $G$ has a source node $\sigma \in V$ and a destination $\tau \in V$. Each node $\alpha \in V$ represent a VNF that has a processing demand $p(\alpha)$. Without loss of generality, we assume that $p(\sigma) = p(\tau) = 0$. Each edge $e \in E$ has a bandwidth demand $b(e)$.

**Placement costs:** There is a non-negative cost $c(\alpha, v)$ for placing the VNF $\alpha$ in $v$. We assume that $c(\sigma, s) = 0$ and $c(\sigma, v) = \infty$, for every $v \neq s$. Similarly, we assume that $c(\tau, t) = 0$ and $c(\tau, v) = \infty$, for every $v \neq t$.

A solution consists of the following:

**Virtual path:** A path from $s$ to $t$ in $G$, namely a sequence of vertices $\alpha_0, \ldots, \alpha_q$, where $\alpha_0 = \sigma$, $\alpha_q = \tau$, and $(\alpha_j, \alpha_{j+1}) \in E$, for every $j \in \{0, \ldots, q-1\}$.

**Physical path:** A simple path from $s$ to $t$ in $G$, namely a sequence of nodes $v_0, \ldots, v_k$, where $v_0 = s$, $v_k = t$, and $(v_i, v_{i+1}) \in E$, for every $i \in \{0, \ldots, k-1\}$.

**Placement:** A function $f$ that maps a virtual path to a simple physical path. Formally, a placement is a function $f : \{\alpha_0, \ldots, \alpha_q\} \rightarrow \{v_0, \ldots, v_k\}$ where

1. $v_i \neq v_{i'}$ if $i \neq i'$.
2. If $f(\alpha_j) = v_i$ and $f(\alpha_{j+1}) = v_{i'}$, then $i \leq i'$.
3. $b(\alpha_j, \alpha_{j+1}) \leq b(v_i, v_{i+1})$, for every $(v_i, v_{i+1}) \in \hat{f}(\alpha_j, \alpha_{j+1})$, where $\hat{f}(\alpha_j, \alpha_{j+1})$ to be the set of physical edges that correspond to it, i.e.,

$$\hat{f}(\alpha_j, \alpha_{j+1}) \triangleq \{(v_i, v_{i+1}) : i' \leq i < i''\},$$

where $f(\alpha_j) = v_i$ and $f(\alpha_{j+1}) = v_{i'}$. 

23
Figure 3.1: An example of an SCP instance and a solution. The DAG on the left is the virtual DAG, and the numbers on its vertices and edges are their demands. The DAG on the right is the physical DAG, and the numbers of its vertices and edges are their capacities. The dashed blue line on the left is the chosen virtual path, while the dashed blue line on the right is the chosen physical path. Dotted lines are used to represent the allocation of each set of virtual vertices to each physical vertex along the path.

4. \[ \sum_{\alpha \in f^{-1}(v)} p(\alpha) \leq p(v) , \] where \( f^{-1}(v) \triangleq \{ \alpha : f(\alpha) = v \} \).

An example of an SCP instance and a solution is given in Figure 3.1. The cost of a placement \( f \) is defined as:

\[
c(f) \triangleq \sum_{j} c(\alpha_j, f(\alpha_j)) .
\]

In SCP the goal is to find a feasible placement of a virtual path into the physical DAG that minimizes the cost.

We also consider an extended version of SCP in which each physical link causes a delay and there is an upper bound \( L \) on the total delay. More formally, each pair of a virtual edge \( \varepsilon \) and physical edge \( e \) is associated with a latency \( \ell(\varepsilon, e) \). Given a placement \( f : \{\alpha_0, \ldots, \alpha_q\} \rightarrow \{v_0, \ldots, v_k\} \), the total latency of the solution is given by

\[
L(f) = \sum_{j=0}^{q-1} \sum_{e \in f((\alpha_j, \alpha_{j+1}))} \ell((\alpha_j, \alpha_{j+1}), e) .
\]

In this case there is an additional constraint that \( L(f) \leq L \).
3.2.2 Notation

Given a virtual DAG \( \mathcal{G} \), we assume the existence of a topological sorting of the vertices and write \( \alpha \prec \beta \) if \( \alpha \) precedes \( \beta \) in this ordering. If the physical network is a DAG, then we make a similar assumption and write \( v \prec u \) if \( v \) precedes \( u \) in this ordering.

3.3 Hardness Results

In this section we present three hardness results. First, we show that even the feasibility question of SCP is NP-hard, and therefore no approximation algorithm exists for SCP. In addition, we show that SCP is NP-hard even if both the physical network and the virtual DAG are simple paths.

We start by showing that even finding a feasible solution is NP-hard.

**Theorem 7. Feasibility of SCP is NP-hard**

*Proof.* We use a reduction from Hamiltonian Path that is known to be NP-hard [49].

Given an instance \( H \) of Hamiltonian Path we construct an instance of SCP. For the physical network we have that \( G \), where \( V(G) = V(H) \cup \{s, t\} \), and \( E(G) = E(H) \cup \{(s, v), (v, t) : v \in V(H)\} \). In addition, \( p(v) = 1 \), for every \( v \), and \( b(e) = 1 \), for every edge. The virtual DAG \( \mathcal{G} \) is a path containing \( n + 2 \) virtual functions, \( \sigma = \alpha_0, \alpha_1, \ldots, \alpha_n, \alpha_{n+1} = \tau \), where \( p(\alpha_i) = 1 \), for every \( i \in \{1, \ldots, n\} \). Also, \( b(\alpha_i, \alpha_{i+1}) = 1 \), for every \( i \).

The construction can clearly be computed in polynomial time. An Hamiltonian Path in \( G \), induces a SCP solution that follows the path, i.e., \( \alpha_i \) is placed in the \( i \)-th vertex along the path. On the other hand, since all demands and capacities are 1, no two VNFs can share a physical node. Hence a SCP solution induces an Hamiltonian path.

Next, we show that SCP is NP-hard even if the physical network is very simple.

**Theorem 8. SCP is NP-hard, even if \( |V \setminus \{s, t\}| = 1 \).**

*Proof.* We prove the theorem using a reduction from PARTITION. Given a PARTITION instance \( \{a_1, \ldots, a_n\} \), we construct an SCP instance as follows. The physical network \( G \) contains three nodes: \( s, v, \) and \( t \), and there are two edges \((s, v)\) and \((v, t)\). The capacity of \( v \) is \( p(v) = \frac{1}{2} \sum_i a_i \). Edge bandwidths are zero. The virtual DAG is composed of \( 2n + 2 \) vertices, namely

\[ V = \{\sigma, \tau\} \cup \{\alpha_1, \ldots, \alpha_n\} \cup \{\beta_1, \ldots, \beta_n\} . \]

Also,

\[ E = \bigcup_i \{\alpha_i, \beta_i\} \times \{\alpha_{i+1}, \beta_{i+1}\} \cup \{(\sigma, \alpha_1), (\sigma, \beta_1), (\alpha_n, t), (\beta_n, t)\} . \]

The DAG is shown in Figure 3.2. The demands are \( p(\alpha_i) = a_i \) and \( p(\beta_i) = 0 \), for every \( i \). Also, \( b(\varepsilon) = 0 \), for every \( \varepsilon \in E \). The costs are: \( c(\alpha_i, v) = 0 \) and \( c(\beta_i, v) = a_i \), for every \( i \).

The SCP instance can be computed in polynomial time. Furthermore, observe that any service chain must be of the form \( \sigma, \gamma_1, \ldots, \gamma_n, \tau \), where \( \gamma_i \in \{\alpha_i, \beta_i\} \). By the construction, we have that the total processing demands plus the total cost is \( \sum_i a_i \). Since \( p(v) = \frac{1}{2} \sum a_i \), we have that the total cost is at least \( \sum_i a_i \). Hence, it is not hard to verify that \( \{a_1, \ldots, a_n\} \in \text{PARTITION} \) if and only if there exists a solution whose cost is \( \frac{1}{2} \sum a_i \). \[ \square \]
Figure 3.2: The specification defined in the proof of Theorem

Figure 3.3: A reduction from a partition instance \(a_1, \ldots, a_n\). The cost function enforces each \(a_i\) to be placed either in \(v_i^-\) or in \(v_i^+\). In the former case this placement incurs no cost but additional latency of \(a_i\), in the later case there will be additional \(a_i\) cost with zero latency.

Next, we show that SCP is NP-hard even if both the physical network and the VNF specification are paths.

**Theorem 9.** SCP is NP-hard, even if both the physical network and the virtual DAG are paths.

Proof. We prove the theorem using a reduction from PARTITION. Given a PARTITION instance \(\{a_1, \ldots, a_n\}\), we construct a virtual path \(\sigma, \alpha_1, \ldots, \alpha_n, \tau\) and a physical path \(s, v_1^-, v_1^+, \ldots, v_n^-, v_n^+, t\). We set \(p(v) = 1\), for every node \(v \neq s, t\), and \(p(a_i) = 1\), for every \(i\). In addition, we set \(b(e) = 1\), for every edge \(e \in E\), and \(b(\epsilon) = 1\), for every edge \(\epsilon \in E\).

As for the costs, we define \(c(\alpha_i, v_i^-) = 0\), \(c(\alpha_i, v_i^+) = a_i\), and for any \(v \not\in \{v_i^-, v_i^+\}\) we set \(c(\alpha_i, v) = \infty\). We also define \(\ell((\alpha_i, a_{i+1}), (v_i^-, v_i^+)) = a_i\), and set \(\ell(\epsilon, e) = 0\), otherwise. Finally, we set \(L = \frac{1}{2} \sum_i a_i\). Figure 3.3 depicts the above reduction. One can verify that \(a_i\) is either counted in the latency or in the cost. Hence, \(\{a_1, \ldots, a_n\} \in \text{PARTITION}\) if and only if there is a placement with cost \(\frac{1}{2} \sum_i a_i\). □

### 3.4 Algorithms for Physical Directed Acyclic Graphs

In this section we present an FPTAS for SCP in DAGs which is based on a dynamic programming algorithm. The algorithm is described in a top down manner. We first assume that costs are polynomially bounded and design a dynamic programming algorithm that computes a minimum cost placement of a virtual path within a single physical node. Next, we provide a dynamic programming algorithm for SCP without the latency constraint. Then, we give an algorithm that copes with a global latency constraint. At the end of the
section we drop our assumption on the costs and use standard scaling techniques to obtain an FPTAS for SCP in DAGs.

### 3.4.1 Placing a Sub-chain in a Physical Node

Assume that we want to place a minimum cost virtual path from a VNF \( \alpha \) to a VNF \( \beta \) within a physical node \( v \). A sequence of VNFs \( \alpha = \alpha_0, \alpha_2, \ldots, \alpha_q = \beta \) is a candidate path if \( (\alpha_i, \alpha_{i+1}) \in \mathcal{E} \), for every \( i \), and \( \sum_i p(\alpha_i) \leq p(v) \). We would like to find such a path with minimum cost, and we denote the cost of such a path by \( \text{cost}^{v}(\alpha \leadsto \beta) \).

We use dynamic programming in order to compute such a path. Let \( \text{PRC}^{v}(\alpha \leadsto \beta, c) \) be the minimum amount of processing required to place a virtual path from \( \alpha \) to \( \beta \) into \( v \) among paths whose cost is at most \( c \). The value of \( \text{PRC}^{v}(\alpha \leadsto \beta, c) \) can be computed recursively as follow:

\[
\text{PRC}^{v}(\alpha \leadsto \beta, c) = \begin{cases} 
\infty & \text{if } c(\alpha, v) > c, \\
p(\alpha) & \text{if } c(\alpha, v) \leq c, \\
\min_{(\gamma, \beta) \in \mathcal{E}} \left\{ \text{PRC}^{v}(\alpha \leadsto \gamma, c - c(\beta, v)) + p(\beta) \right\} & \text{otherwise}.
\end{cases}
\]

Observe that if \( c(\alpha, v) > c \), then a placement is not possible with a budget \( c \). Otherwise, if \( \alpha = \beta \), then the best path is the one containing only \( \alpha \). If \( \alpha \neq \beta \), then an optimal path ends with an edge \( (\gamma, \beta) \in \mathcal{E} \), which means that the processing placed at \( v \) consists of \( p(\beta) \) plus the minimum amount of processing of a path from \( \alpha \) to \( \gamma \), whose cost is at most \( c - c(\beta, v) \).

To complete our argument observe that the following holds:

\[
\text{cost}^{v}(\alpha \leadsto \beta) = \min \left\{ c : \text{PRC}^{v}(\alpha \leadsto \beta, c) \leq p(v) \right\}.
\]

Since the costs are assumed to be polynomially bounded, there is a polynomial number of states to be computed, and the computation of each of them can be done in linear time. Hence, the total running time is polynomial. Finally, we note that the above algorithm computes the minimum amount of processing, but may also be used to compute the actual placement that achieves this value using standard techniques.

### 3.4.2 Placing a Service Chain

In this section we describe a dynamic programming algorithm for placing a service chain without a global latency bound.

Consider an optimal solution of SCP, i.e., a service chain from \( \sigma \) to \( \tau \) which is placed along a path from \( s \) to \( t \) in the physical DAG. Suppose \( v \) is a node along the path from \( s \) to \( t \). Also, let \( \alpha \) be the last VNF in the service chain which is placed in the path from \( s \) to \( v \). It must be that the placement of the VNFs from \( \sigma \) to \( \alpha \) in the path from \( s \) to \( v \) is the best one among all placements of a virtual path from \( \sigma \) to \( \alpha \) in a path from \( s \) to \( v \). In other words, any partial placement of an optimal placement is also optimal. Our algorithm is based on this property.

We define a state for each pair of VNF \( \alpha \) and physical node \( v \). Let \( \text{cost}(\alpha, v) \) stand for the minimum cost placement of a virtual path from \( \sigma \) to \( \alpha \) in a path from \( s \) to \( v \), where \( \alpha \) is the last VNF which is placed along the physical path. In addition, we write \( (u, v) \rightarrow (\gamma, \beta) \) if \( v \) is reachable from \( u \) using only edges with bandwidth at least \( b(\gamma, \beta) \), namely if there
Figure 3.4: The optimal placement of a path to $\alpha$ into a path to $v$ can be efficiently computed by breaking the problem into the problem of placing a path to $\gamma$-path into a path to $u$ (blue, dashed rectangles) and the problem of embedding a path from $\beta$ to $\alpha$ into $v$ (orange, dotted). The path from $u$ to $v$ must carry at least $b(\gamma, \beta)$ bandwidth.

is a path from $u$ to $v$ such that the bandwidth of all edges in the path is at least $b(\gamma, \beta)$. $\text{cost}(\alpha, v)$ can be computed recursively, as follows:

$$\text{cost}(\alpha, v) = \begin{cases} 0 & \alpha = \sigma, v = s, \\ \min_{\beta \prec \alpha, (\gamma, \beta) \in \mathcal{E}, u \prec v, (u, v) \not\rightarrow (\gamma, \beta)} \{\text{cost}(\gamma, u) + \text{cost}_v(\beta \rightarrow \alpha)\} & \text{otherwise}. \end{cases}$$

The desired value is $\text{cost}(\tau, t)$. Consider an optimal placement of a virtual path ending at $\alpha$ within a path ending at $v$. Let $\gamma$ be the first VNF along the virtual path that is not placed in $v$. Also, let $u$ be the node in which $\gamma$ is placed. Hence, the optimal placement is composed of three segments: an optimal placement of a virtual path that ends in $\gamma$ in a physical path ending at $u$, a virtual edge $(\gamma, \beta)$ is placed in the path from $u$ to $v$, and a minimum cost placement of a virtual path from $\beta$ to $\alpha$ in $v$, that does not violate the capacity of $v$. Thus, we check all pairs $(\gamma, u)$, where $\gamma \prec \alpha$ and $u \prec v$, and for each pair we consider any neighbor $(\gamma, \beta) \in \mathcal{E}$ and $\beta \prec \alpha$ such that $(u, v) \not\rightarrow (\gamma, \beta)$. The recursive computation is illustrated in Figure 3.4.

As for the running time, observe that checking whether $(u, v) \not\rightarrow (\gamma, \beta)$ can be done using DFS in linear time. Since the number of state is polynomial, and each state can be computed in polynomial time, the total running time is polynomial. Finally, we note that the above algorithm computes the minimum amount of cost, but may also be used to compute the actual placement that achieves this value using standard techniques.

### 3.4.3 Placing a Service Chain with a Latency Bound

We now consider the SCP problem with latency. Recall that in this variant of the problem we are also given a latency function $\ell: E \times \mathcal{E} \rightarrow \mathbb{R}_+$, and a latency upper bound $L$. The goal is to find a minimum cost placement that also respect the latency constraint.
Let \( \text{LAT}(\alpha, v, c) \) be the minimum latency created by a placement of a virtual path from \( \sigma \) to \( \alpha \) in a path from \( s \) to \( v \) whose cost is at most \( c \). Also, let \( \text{LAT}(\gamma, \beta, w \rightarrow v) \) be the minimum possible latency of path from \( w \) to \( v \) such that the bandwidth of all edges in the path is at least \( b(\gamma, \beta) \). That is,

\[
\text{LAT}(\gamma, \beta, w \rightarrow v) = \min_{f: b(\gamma, \beta) \leq \min_{e \in f(\gamma, \beta)} b(e)} \sum_{e \in f(\gamma, \beta)} \ell((\gamma, \beta), e).
\]

Observe that \( \text{LAT}(\gamma, \beta, w \rightarrow v) \) can be computed using a Shortest Path algorithm, were we consider only edges whose bandwidth cap is at least \( b(\gamma, \beta) \). Now we are ready for the computation of \( \text{LAT}(\alpha, v, c) \):

\[
\text{LAT}(\alpha, v, c) = \begin{cases} 
\infty & \text{if } c < 0, \\
0 & \text{if } c = 0, \alpha = \sigma, v = s, \text{ and } c \geq 0, \\
\min_{\beta < \alpha, (\gamma, \beta) \in E, w < c, (u, w) \in E, (u, v) \rightarrow (\gamma, \beta)} \left\{ \begin{array}{l}
\text{LAT}(\gamma, \beta, w \rightarrow v) + \\
\text{LAT}(\gamma, u, c - \text{COST}_v(\beta \rightarrow \alpha))
\end{array} \right\} & \text{otherwise.}
\end{cases}
\]

The minimum latency of a placement of a path to \( \alpha \) in a path to \( v \) whose cost is at most \( c \) can be divided into two values: the latency caused by the placement of a path to \( \gamma \), where \( (\gamma, \beta) \in E \), and \( \beta < \alpha \), with cost \( c - \text{COST}_v(\beta \rightarrow \alpha) \), and the latency caused by placing a virtual edge \( (\gamma, \beta) \) on a physical path from \( w \) to \( v \), where \( w < v \) and \( (u, w) \in E \). The optimal value is \( \min \{ c : \text{LAT}(\tau, t, c) \leq L \} \).

Since the computation of \( \text{LAT}(\gamma, \beta, w \rightarrow v) \) can be done efficiently, the computation for a triple \( (\alpha, v, c) \) can be done in polynomial time. Hence, the total running time is polynomial. Finally, as in the previous algorithms, one may use the algorithm to compute a corresponding placement using standard techniques.

### 3.4.4 FPTAS for General Costs

In this section we present an FPTAS for SCP in DAGs for general costs, namely without the assumption that costs are polynomially bounded. Our algorithm is similar to the FPTAS for the MINIMUM KNAPSACK problem (see [65][Chapter 13] and references therein).

Let \( f \) be an optimal placement, and let \( c_{\text{max}} \) be the maximum cost of a VNF placement by \( f \), i.e., \( c_{\text{max}} = \max_{f(\alpha) = u} c(\alpha, v) \). Given a constant \( \varepsilon > 0 \), we define

\[
c'(\alpha, v) = \begin{cases} 
\frac{c(\alpha, v)}{c_{\text{max}}} & \text{if } c(\alpha, v) \leq c_{\text{max}}, \\
\infty & \text{if } c(\alpha, v) > c_{\text{max}}.
\end{cases}
\]

Let \( f' \) be an optimal placement with respect to \( c' \).

**Lemma 14.** \( c(f') \leq (1 + \varepsilon)c(f) \).
Proof. We have that
\[ c(f') = \sum_{\alpha} c(\alpha, f'(\alpha)) \leq \frac{\varepsilon c_{\text{max}}}{n} \cdot \sum_{\alpha} c'(\alpha, f'(\alpha)) \]
\[ \leq \frac{\varepsilon c_{\text{max}}}{n} \cdot \sum_{\alpha} c'(\alpha, f(\alpha)) \]
\[ \leq \frac{\varepsilon c_{\text{max}}}{n} \cdot \sum_{\alpha} \left( \frac{c(\alpha, f(\alpha))}{c_{\text{max}}} \cdot \frac{n}{\varepsilon} + 1 \right) \]
\[ = c(f) + \varepsilon c_{\text{max}} \]
\[ \leq (1 + \varepsilon)c(f) , \]
where the second inequality is due to the optimality of \( f' \) with respect to \( c' \).

This leads to the following result:

**Theorem 10.** There exists an FPTAS for SCP in DAGs.

**Proof.** First, observe that there are \( |V| \cdot |V| \) candidates for the value of \( c_{\text{max}} \), one for each pair of virtual function and physical vertex. Hence, given \( \varepsilon > 0 \), we run the dynamic programming algorithm from Section 3.4.3 \(|V|\cdot|V|\) times, once for each possible value of \( c_{\text{max}} \), and choose the best placement. According to Lemma 14 the placement that was computed for the value of \( c_{\text{max}} \) that corresponds to an optimal solution is \((1 + \varepsilon)\)-approximate. Hence, the best placement is also \((1 + \varepsilon)\)-approximate. The running time is polynomial in the input size and in \( 1/\varepsilon \).

3.5 General Networks

In this section we consider SCP when the physical network is an undirected graph. Recall that even the feasibility version of SCP is NP-hard in general networks, and therefore we focus on a special case. A vertex \( v \in V \) is called neighborly if it has more than two neighbors. First, we assume that there exists an optimal placement whose physical path consists of at most \( k \) neighborly vertices. For example, this can be assumed if all simple paths from \( s \) to \( t \) contain at most \( k \) neighborly vertices. In this case we present a randomized algorithm that computes an \((1 + \varepsilon)\)-approximate placement with high probability whose running time is \( O(k! \cdot \text{poly}(n)) \). Our second algorithm works under the stronger assumption that there are \( k \) neighborly vertices in the network. In this case, we present a deterministic algorithm whose running time is \( O(k! \cdot \text{poly}(n)) \).

Before presenting our algorithms we assume without loss of generality that each vertex in the network is found on a path from \( s \) to \( t \), since otherwise it can be eliminated. In particular, it follows that the degree of any vertex, besides maybe \( s \) and \( t \), is at least 2. In addition this implies that the network is connected.

3.5.1 Randomized Algorithm

Our randomized algorithm consists of \( k! \) iterations of the following two phases: an orientation phase that applies a random orientation to the physical network, and an execution of the FPTAS for DAGs given in Theorem 10. The algorithm finds a \((1+\varepsilon)\)-approximate placement
with probability \((1 - 1/e)\), and the running time of the algorithm is \(O((n + t(n))k!))\), where \(t(n)\) is the time it takes to compute a placement when the physical network is a DAG. As usual, one may amplify the probability of success using repetition.

The orientation phase is done as follows. Let \(N\) be the set of neighborly vertices. We assume without loss of generality that \(N = \{1, \ldots, |N|\}\). Let \(\pi : N \to N\) be a random permutation. Define \(\pi(s) = 0\), and \(\pi(t) = N + 1\). We direct the edges according to \(\pi\).

Observe that each edge \(e \in E\) is found on a simple path between two neighborly nodes \(v_e\) and \(v'_e\), between \(v_e = s\) and a neighborly node \(v'_e\), or between neighborly node \(v_e\) and \(v_e = t\). In this path all internal vertices are in \(V \setminus N\). The edge \(e\) is directed towards \(v_e\), if \(\pi(v_e) > \pi(v'_e)\), and otherwise it is directed towards \(v'_e\). Observe that when this process terminates we get a DAG, denoted by \(G_\pi\), where there is a topological order than is consistent which \(\pi\). Figure 3.5 depicts the orientation phase. A naïve implementation of this phase would run in \(O(|V| |E|)\) time.

Upon completing the orientation, we use our previous algorithm to find a placement. We repeat this process \(k!\) times, each time with a new independent random permutation, and keep the best embedding so far.

**Theorem 11.** There exists an algorithm, that given an SCP instance with an optimal solution that contains at most \(k\) neighborly vertices, finds a \((1 + \varepsilon)\)-approximate placement with high probability, whose running time is \(O(k! \cdot \text{poly}(n))\).

**Proof.** Consider an optimal placement with at most \(k\) neighborly vertices whose set is denoted by \(N'\), and let \(\pi'\) be the permutation it induces on the neighborly vertices in the physical path. If the random permutation \(\pi\) orders the neighborly vertices in \(N'\) correctly, i.e., if \(\pi(v) = \pi'(v)\), for every \(v \in N'\), then the optimal placement is feasible with respect to \(G_\pi\). Hence, the FPTAS will find a \((1 + \varepsilon)\)-approximate placement in \(G_\pi\). This placement is feasible, and therefore \((1 + \varepsilon)\)-approximate placement, with respect to \(G\).
The permutation $\pi$ agrees with $\pi'$ with probability $1/|N'|! \geq 1/k!$, and thus the probability that $\pi$ agrees with $\pi'$ in $k!$ independent iterations is at least $1 - (1 - 1/k!)^{k!} \geq 1 - e^{-1}$. One may amplify the success probability to $1 - \frac{1}{n}$ using $O(\log n)$ repetitions: $1 - (1 - 1/k!)^{\ln n \cdot k!} \geq 1 - e^{-\ln n} = 1 - 1/n$.

### 3.5.2 Fixed Parameter Algorithm

In the case where $|N| = k$ we can simply examine the $k!$ permutations of the neighborly vertices. For each possible permutation the algorithm applies the graph orientation procedure as described above, and then executes the FPTAS for DAGs given in Section 3.4.4. Thus the running time of the algorithm is $O(k! \cdot \text{poly}(n))$. We note that permutation enumeration for $k$ numbers takes $O(k!)$ time (see e.g., [41, 103]). The result is a fixed parameter approximation algorithm, where the parameter is the number of neighborly vertices.

**Theorem 12.** There exists an $O(k! \cdot \text{poly}(n))$-time algorithm, that given an SCP instance with $k$ neighborly vertices, finds a $(1 + \varepsilon)$-approximate placement.

Next we show that there are graphs for which the running time can be decreased. More specifically, we propose a different method to apply an orientation to an undirected graph. While the proposed method does not surpass the above mentioned orientation in the worse case, in certain cases, the new method yields an exponential improvement.

Given a connected graph $G$, a bi-connected component (or a block) in $G$ is a maximal bi-connected subgraph of $G$. Any connected graph can be decomposed into bi-connected subgraphs that form a tree called a block-cut tree. In this tree every vertex is a block of $G$ and two blocks are attached if they share a vertex, which is called a cut-vertex. Computing such a tree can be done in linear time [60].

The orientation works as follow. Given an undirected graph $G = (V, E)$, we first compute its block-cut tree. Then on the unique path from $s$ to $t$ in the block-cut tree we internally orient each block arbitrarily to produce a DAG. Note that if $B_1, \ldots, B_\ell$ are the blocks of $G$ on the path from $s$ to $t$ then the first and last vertices with respect to each block are known and only the order of the internal neighborly vertices in each block is needed to be determined, thus, we can find all relevant orientations by repeating the process above $\Pi_i=1 |N_i|$ times, where $N_i$ is the set of internal neighborly vertices in $B_i$. See Figure 3.6 for an example.

This leads to the following result.

**Theorem 13.** There exists an $O(\Pi_i=1 |N_i| \cdot \text{poly}(n))$-time algorithm, that given an SCP instance, computes a $(1 + \varepsilon)$-approximate placement.

We also note that if the given graph is a cactus graph then there is only one possible orientation, and thus this approach results in a polynomial time algorithm.

**Theorem 14.** There exists a polynomial-time algorithm, that given an SCP instance where the network is a cactus graph, computes a $(1 + \varepsilon)$-approximate placement.

We note that cactus graphs were also considered in [47].

### 3.6 Fault Tolerance

In this section we consider the problem of a Service Chain Placement when there is also a need for fault tolerance. Note that we consider the problem without latency.
Formally, given a physical network, a virtual specification, and a parameter \( r \in \mathbb{N} \), an \( r \)-robust placement is a collection of \( r \) placements, \( f_1, \ldots, f_r \), that correspond to \( r \) internal vertex disjoint physical paths. In the fault tolerant version of SCP we are given \( r \) costs functions \( c_1, \ldots, c_r \), and we are looking for an \( r \)-robust placement that minimize the total cost:

\[
\sum_{i=1}^{r} c_i(f_i) = \sum_{i=1}^{r} \sum_{j} c_i(\alpha_j, f_i(\alpha_j)) .
\]

We refer to this problem as \( r \)-robust SCP. The motivation for using different costs functions is that it may be the case that the first placement is the primal placement that may be replaced by one of the other placements in case of a malfunction along the physical path that is used by \( f_1 \). Hence, the cost function \( c_2 \) may take into account the probability of using the placement \( f_2 \).

In this section we study the problem of efficiently finding an optimal 2-robust placement. As in Section 3.4 we obtain an FPTAS by first assuming that costs are polynomially bounded. We note that the algorithm presented here can be generalized to find an optimal \( r \)-robust placement for any constant \( r \).

\[\text{Two paths are internal vertex disjoint is they do not share a vertex apart from, maybe, the first and last vertices on the paths.}\]
Before presenting our algorithm, we show that using the algorithm from Section 3.4 repeatedly, \( r \) times, may result in failure. In this approach, we find an optimal placement with respect to \( c_i \) and then remove all the vertices that are used by the computed placement, before computing an optimal placement with respect to \( c_{i+1} \). Consider the following example.

The network, source, and destination are depicted in Figure 3.7. The virtual DAG is a path consisting of two virtual functions \( \alpha \) and \( \beta \). Assume that the latencies, bandwidths, and processing time of the request are negligible. Also, assume that \( c_1(1, \alpha) = c_1(2, \beta) = 1 - \varepsilon \) and \( c_1(1, \beta) = c_1(2, \alpha) = 1 \). The above algorithm is unable to find a 2-robust placement in this case. After the first iteration there is no path from \( s \) to \( t \), although there is a 2-robust placement that costs \( 4 - 2\varepsilon \).

### 3.6.1 Layered Graph Transformation

As a first step the input DAG is transformed into a *layered graph*. A directed acyclic graph \( G = (V, E) \) is called *layered* if there is a mapping \( h : V \rightarrow \mathbb{N} \), such that if \( (v, u) \in E \) then \( h(u) = h(v) + 1 \). That is, each vertex \( v \) belongs to the layer \( h(v) \), and each edge goes from one layer to the next. We denote the \( i \)th layer by \( L_i \), i.e., \( L_i = \{ v : h(v) = i \} \). In a layer \( \text{DAG} \) we have that \( E \subseteq \bigcup_i (L_i \times L_{i+1}) \).

Given a directed acyclic graph \( G = (V, E) \), let \( \pi : V \rightarrow [n] \) be an arbitrary topological ordering of the vertices in \( G \). We transform \( G \) into a layered DAG \( G' \) by replacing every edge \( (v, u) \in E \) with a directed path of length (number of edges) \( \pi(u) - \pi(v) \) from \( u \) to \( v \). Figure 3.8 depicts this process. It is straightforward to see that the resulting graph \( G' \) is indeed a layered DAG.

We set the processing capacity of each internal vertex along a path that was added to be zero, and the bandwidth of the arcs along the path to be \( b(v, u) \). Observe that the transformation can be done in polynomial time and that every placement on \( G \) corresponds to a placement on \( G' \) with the same cost, and vice versa.

### 3.6.2 Fault Tolerance on Layered Graphs

In this section we assume that the input graph is a layered graph, i.e., \( G = (\bigcup_{i=1}^f L_i, E) \), where \( E \subseteq \bigcup_{i=1}^{f-1} (L_i \times L_{i+1}) \). We assume that the number of vertices in each layer is at least two, or otherwise a 2-robust placement does not exist. Also, we add a dummy virtual edge \((\tau, \tau')\) to the virtual graph \( \mathcal{G} \), where \( p(\tau') = 0 \) and \( c(\tau', v) = \infty \) for every \( v \neq t \).

The algorithm uses dynamic programming. We find the optimal 2-robust placement for every combination of two physical vertices from the same layer, and every combination of two virtual edges. Formally, we define \( FT((\alpha_1, \zeta_1), (\alpha_2, \zeta_2), v_1, v_2) \) to be the minimum cost
2-robust placement of two service chains that start at $\sigma$ and end with the edges $(\alpha_1, \zeta_1)$ and $(\alpha_2, \zeta_2)$, respectively, such that a virtual paths from $\sigma$ to $\alpha_1$ and $\alpha_2$ are placed into internal vertex disjoint physical paths that start at $s$ and end at $v_1$ and $v_2$, respectively. Recall that $\text{cost}_j^l(\gamma_j, \beta_j)$ is the cost, with respect to $c_j$, of a minimum cost path from $\gamma_j$ to $\beta_j$ whose total processing requirement can be handled by $v_j$, where $j \in \{1, 2\}$. Consider the following recursive definition. First,

$$\text{FT}((\alpha_1, \zeta_1), (\alpha_2, \zeta_2), s, s) = \text{cost}_s^1(\sigma \leadsto \alpha_1) + \text{cost}_s^2(\sigma \leadsto \alpha_2).$$

Otherwise, for $v_1 \neq v_2$ of $v_1 = v_2 = t$, such that $v_1, v_2 \in L_i$, for some $i$, we have

$$\text{FT}((\alpha_1, \zeta_1), (\alpha_2, \zeta_2), v_1, v_2) = \min \left\{ \text{FT}((\gamma_1, \beta_1), (\gamma_2, \beta_2), u_1, u_2) + \text{cost}_{v_1}^1(\beta_1 \leadsto \alpha_1) + \text{cost}_{v_2}^2(\beta_2 \leadsto \alpha_2) \right\}$$

where we take the minimum over all the possible virtual functions $\gamma_1, \beta_1, \gamma_2, \beta_2$ and physical nodes $u_1, u_2$, such that:

- $\gamma_j \preceq \alpha_j$,
- $\beta_j \preceq \alpha_j$ or $\beta_j = \zeta_j$,
- $(\gamma_j, \beta_j) \in \mathcal{E}$,
- $u_1 \neq u_2$ or $u_1 = u_2 = s$.

$^2$Note that if $\beta_j = \zeta_j$, then $\text{cost}_{v_j}^l(\gamma_j \rightarrow \alpha_j) = 0.$
• $(u_j, v_j) \in E$, and

• $b(\gamma_j, \beta_j) \leq b(u_j, v_j)$

The optimal 2-robust placement is obtained by the call $FT((\tau, \tau'), (\tau', t), t, t)$

If all costs are polynomially bounded, then computation of $FT((\alpha_1, \xi_1), (\alpha_2, \xi_2), v_1, v_2)$ can be done in polynomial time, since the minimum is taken over a polynomial number of candidates and the computation of $\text{cost}^t_{\gamma_j}(\gamma_j, \beta_j)$ can be done in polynomial time, as shown in Section 3.4.2. Hence, the overall running time is polynomial. The above algorithm computes the cost of an optimal 2-robust placement, and as usual, one may compute an optimal 2-robust placement using standard techniques.

Using the same approach as was used in Section 3.4.4 we obtain the following result.

**Theorem 15.** There exists an FPTAS for 2-robust SCP (without latency) in DAGs.

Finally, we note that the above algorithm does not take into account the latency constraint. It is possible, however, given a latency constraint, $L$, to modify the algorithm to return an optimal solution such that the total latency of the two placements is no more than $2L$. This is done in a manner which is similar to what was done in Section 3.4.3, but with an upper bound on the total latency of both paths. Such a solution is guaranteed to be optimal in terms of costs. Also, if the total latency is $2L$, then at least one of the placements has lower latency than $L$ and the other one has latency that is at most $2L$. 

36
Chapter 4

Monotone Submodular Function Maximization

4.1 Introduction

In this thesis we consider the problems of maximizing a monotone\footnote{1} submodular function given a knapsack constraint. In this problem we are given a ground set $U$ of size $n$, a monotone submodular function $f : 2^U \rightarrow \mathbb{R}_+$, a cost function $c : U \rightarrow \mathbb{R}_+$, and a budget $\beta$. The goal is to find a subset of elements $S$ that maximizes $f(S)$ such that the total cost of the elements in $S$ does not exceed the budget, i.e., $\sum_{x \in S} c(x) \leq \beta$. For abbreviation we denote this problem by \textsc{Submodular Knapsack}. Besides being a natural problem on its own right, capturing the classic Knapsack problem, \textsc{Submodular Knapsack} admits practical applications, e.g., entropy maximization in graphical models \cite{75}, and document summarization \cite{87}. In this work we aim to find fast and simple algorithms for the \textsc{Submodular Knapsack} problem.

We assume the standard value oracle model, where the algorithm can access the objective $f$ via queries of the form: “what is $f(S)$?” for every $S \subseteq U$. The running time is the total number of value oracle queries and numerical operations performed by the algorithm. In all our algorithms the former dominates the later, and thus we are satisfied with counting the number of oracle queries alone.

Building upon the work of Khuller et. al. \cite{70}, Sviridenko \cite{106} presented a tight approximation of $(1 - 1/e)^2 \approx 0.632$ for \textsc{Submodular Knapsack}. The algorithm of \cite{106} returns the best of all subsets of $U$ of size at most three, where each subset of size three is greedily extended by the standard greedy rule that maximizes the “bang per buck”. This results in an impractical algorithm whose running time is $O(n^5)$. A fast algorithm whose running time is just that of the greedy algorithm, i.e., $O(n^2)$, was given by \cite{70} and it achieves a worse approximation of $(1 - e^{-1/2}) \approx 0.393$\footnote{2}.

Deviating from the above combinatorial approach of \cite{70, 106} to \textsc{Submodular Knapsack}, Badanidiyuru and Vondrák \cite{15} initiated a different line of research based on both continuous and discrete techniques. They presented an algorithm achieving an approxim-
mation of $(1 - 1/e - \varepsilon)$ whose running time is $O(n^2(\varepsilon^{-1} \log n)^{poly(\varepsilon^{-1})})$, for every constant $\varepsilon > 0$. Building upon the approach of [15], Ene and Nguyễn [35] presented an algorithm achieving the same approximation guarantee whose running time is $O(\varepsilon^{-O(\varepsilon^{-1})} n \log^2 n)$. Both algorithms of [15, 35] are theoretically interesting and appealing, as they require the introduction of novel ideas that enable one to extrapolate between the discrete and continuous approaches. Unfortunately, both are not simple nor fast, as even stated by the authors themselves, e.g., see [35]. To best exemplify the impracticality of these algorithms one needs only to choose $\varepsilon = 1/4$. This results in an approximation of $(1 - 1/e - 1/4)$, which is worse than the fast algorithm of [70] since $(1 - 1/e - 1/4) < (1 - e^{-1/2})$, and an impossible running time of at least $2^{312}n$ [35].

**Our Results**  Our results can be partitioned into two parts: (1) fast and simple combinatorial algorithms for **Submodular Knapsack**; and (2) a general method for “amplifying” any algorithm for **Submodular Knapsack** by improving its approximation factor while losing little in the running time.

**Fast and Simple Combinatorial Algorithms:** We note that the proof that the fast algorithm of [70] (named Modified Greedy by the authors) achieves an approximation guarantee of $(1 - e^{-1/2})$ is incorrect [93]. We rectify this and note that a correct proof requires a different argument than the one appearing in [70]. This is summarized in the following theorem.

**Theorem 16.** The Modified Greedy algorithm of Kuller et. al. [70] for the **Submodular Knapsack** problem achieves an approximation of $(1 - e^{-1/2})$.

Building upon the correct proof of Theorem 16, we present a remarkably simple algorithm that chooses the best between the greedy algorithm and the best pair of elements. Our algorithm retains the same running time of $O(n^2)$ as the greedy algorithm and the Modified Greedy algorithm of [70], but achieves an improved approximation guarantee of $\approx 0.453647$ (whereas the Modified Greedy algorithm provides a guarantee of only $1 - e^{-1/2} \approx 0.393$). In fact, the number of oracle queries used by our algorithm is $3n^2/2 + n$. This is summarized in the following theorem.

**Theorem 17.** The **Submodular Knapsack** problem admits an algorithm that runs in time $O(n^2)$ and achieves an approximation of $0.453647$.

**Amplification:** We present a general method for “amplifying” algorithms for **Submodular Knapsack**. Specifically, given any black box algorithm $A$ that obtains an approximation guarantee of $r$ and a number $k \in \mathbb{N}$ of executions, we show how to obtain a new algorithm for **Submodular Knapsack** with a better approximation than $r$ and a running time that equals $k$ times the running time of the black box algorithm plus an additional $3n^2/2 + n$ oracle queries. This is summarized in the following theorem.

**Theorem 18.** Let $A$ be an algorithm for the **Submodular Knapsack** problem achieving an approximation of $r$, where $r < 1/2$. Let $k \geq \max \left\{ 2, \frac{2^{\log_2 \left( \frac{1}{1-r} \right)}}{\log_2 \left( \frac{1}{1-r} \right)} + 1 \right\}$. Then there exists an uses $3n^2/2 + n$ oracle queries plus $k$ times the running time of $A$ and achieves an approximation of $r^*$, where:

1. $r^* = \max_{0 < \alpha \leq \ln 2} \left\{ \min \left\{ 1 - e^{-\alpha}, D(\alpha) + (1 - r) \left( \frac{2^e(\alpha)}{1 + e(\alpha)} (1 - D(\alpha)) - 1 \right) \right\} \right\}$. 

38
2. \( D(\alpha) = \frac{(1 - e^{-\alpha})}{B(\alpha)}. \)

3. \( \nu(\alpha) = 2^{-\frac{\log A(\alpha)}{4}} - 1. \)

4. \( A(\alpha) = \frac{1}{1 - e^{-\alpha}} - \frac{1}{B(\alpha)}. \)

5. \( B(\alpha) = 1 - e^{-\frac{1}{2}}. \)

Given any black box algorithm \( A \) for \textsc{Submodular Knapsack} Theorem 18 can be used to answer two interesting questions: (1) given an upper on the running time of the algorithm what is the best approximation that can be obtained using our amplification framework? (2) given a target approximation guarantee what is the required running time using our amplification framework? For example, choosing the black box algorithm \( A \) to be the algorithm whose existence is guaranteed by Theorem 17, amplifying it with \( k = 6 \) we can obtain an algorithm achieving an improved approximation of 0.48 and only uses \( 10.5n^2 + 7n \) oracle queries. Moreover, one can repeatedly apply the amplification, each time with a suitable choice of \( k \), obtaining an approximation guarantee that approaches \( \frac{1}{2} \).

**Our Techniques** We adopt a purely combinatorial approach, thus deviating from the recent line of work of [15, 35]. Both our results, an improved fast and simple algorithm as well as the amplification, are inspired by the tight (but slow) algorithm of [70, 106] for \textsc{Submodular Knapsack}. We require two new insights that allow us to obtain our results.

The first insight relates to how algorithms for \textsc{Submodular Knapsack} can be analyzed. At the heart of the analysis of the tight (but slow) algorithm of [70, 106] lies the observation that as long as no element of the optimal solution was dropped by the greedy “bang per buck” algorithm, value is accumulated at a rate that depends on the fraction of the budget that was used so far. Formally, if the greedy “bang per buck” uses a fraction \( \beta \) of the budget and it did not drop any element that belongs to the optimal solution so far, then its value is at least a fraction of \( \frac{1}{1 - e^{-\alpha}} \) of the value of the optimal solution. Once the first element of the optimal solution is dropped, no further analysis of the algorithm is given in [70, 106].

We present an improved approach to analyzing the greedy “bang per buck” algorithm and show that in certain cases value can still be accumulated even after elements of the optimal solution were dropped. This plays a crucial role in all of our results: (1) a correct proof to the fast algorithm of [70] (Theorem 16); (2) an improved fast and simple algorithm (Theorem 17); and (3) our amplification framework (Theorem 18).

The second insight relates to the design of the amplification framework. The tight (but slow) algorithm of [70, 106] needs to find a small (of size at most three) subset of the most valuable elements in the optimal solution, and extend it by the greedy “bang per buck” algorithm. This is done by simple enumeration over all small subsets, thus resulting in a tight (but slow) algorithm that runs in \( O(n^3) \) time. The crucial point in the analysis is that the algorithm needs to find the \textit{exact} subset. Hence, since the optimal solution is not known, enumeration is used over the \( O(n^3) \) such subsets.

We, on the other hand, adopt a different approach where it is enough to find a small subset that is \textit{comparable} in both value and cost to the most valuable small subset in the optimal solution. We show that there is only a \textit{constant} number of such comparable small subsets. This allows us, when combined with the first insight, to devise our method.

\footnote{The greedy “bang per buck” is formally described in Section 4.2}
for amplifying any algorithm for the Submodular Knapsack problem: improving its approximation factor by running it a constant (the parameter $k$ from Theorem 18) number of times on meticulously selected instances.

**Related Work** A simple greedy algorithm which yields an $(1-e^{-1})$ approximation for the special case of Submodular Knapsack with uniform costs (i.e. cardinality constraint), along with a matching lower bound in the oracle model is known since the late 70’s due to the works of Nemhauser, Wolsey and Fisher [95][94]. Their works also achieved a $1/2$ approximation for the problem of maximizing a monotone submodular function subject to a matroid constraint [45]. An optimal $(1-e^{-1})$ approximation for the latter problem was only attained in the 2000’s with the introduction of the continuous greedy and matching pipage rounding [22].

The problem can also be generalized to the problem of maximizing submodular function subject to multiple knapsack constraints for which Kulik et. al. provided a $(1-e^{-1} - \epsilon)$ approximation for every $\epsilon > 0$ [82].

**Chapter Organization** Section 4.2 contains needed notations and description of previous algorithms. Section 4.3 contains a correct proof of the fast algorithm of Khuller et. al. [70], whereas Section 4.4 contains our improved fast algorithm. Lastly, Section 4.5 describes our amplification method.

### 4.2 Preliminaries

First, we start with some notations that will enable us to simplify the presentation of the algorithms and their proofs. We use the notation of $f(B|A)$ to denote the marginal value of $B$ with respect to $A$, i.e., $f(A \cup B) - f(A)$. When it is clear from the context we use $x$, where $x \in U$, to denote the set $\{x\}$. For any subset $A \subseteq U$ we denote by $c(A) = \sum_{x \in A} c(x)$.

Moreover, for an ordered set $A = \{a_1, \ldots, a_k\}$ we use $A_i$ to denote its prefix of size $i$, i.e., $\{a_1, \ldots, a_i\}$ ($A_0$ is set to the empty set). Since $f$ is monotone and non-negative, we can assume without loss of generality that $f(\emptyset) = 0$.

Second, let us present known fast algorithms for Submodular Knapsack. The greedy algorithm for Submodular Knapsack which uses the “bang per buck” greedy rule appears in Algorithm 2. It receives as parameters the ground set $U$, the submodular objective $f$ (via its value oracle), the cost function $c$, and the budget $\beta$.

**Algorithm 2: Greedy$(U, f, c, \beta)$**

1. $S \leftarrow \emptyset$
2. while $U \neq \emptyset$ do
3. $e' \leftarrow \arg \max \{f(e|S)/c(e) : e \in U\}$
4. $U \leftarrow U \setminus \{e'\}$
5. if $c(S \cup \{e'\}) \leq \beta$ then
6. $S \leftarrow S \cup \{e'\}$
7. end
8. end
9. return $S$. 

40
If the condition in line 5 is evaluated to false we say that the algorithm dropped the element $e'$. It is well known that in general the approximation ratio of Algorithm 2 can be arbitrarily bad.

Khuller et. al. [70] presented the Modified Greedy algorithm, which returns the best between the greedy algorithm (Algorithm 2) and the single best element. This appears in Algorithm 3 (its input is identical to that of Algorithm 2).

**Algorithm 3: Modified Greedy** $(U, f, c, \beta)$

1. $S \leftarrow \text{Greedy}(U, f, c, \beta)$
2. $T \leftarrow \arg\max \{S, \arg\max \{f(e) : e \in U, c(e) \leq \beta\}\}$
3. return $T$

Third and last, we present a lemma (can be derived from, e.g., [70]) that enables one to analyze the performance of the greedy algorithm (Algorithm 2).

**Lemma 15.** Let $A = \{a_1, \ldots, a_k\}$ and $B$ be two subsets such that for all $1 \leq i \leq k$ and for all $e \in B$ it holds that $\frac{f(a_i | A_{i-1})}{c(a_i)} \geq \frac{f(e | A_{i-1})}{c(e)}$. Then, $f(A) \geq (1 - e^{-\frac{c(a_k)}{c(B)}}) f(B)$.

**Proof of Lemma 15.** Consider $a_i$ and observe that:

\[
\frac{f(a_i | A_{i-1})}{c(a_i)} c(B) = \sum_{e \in B} \frac{f(a_i | A_{i-1})}{c(a_i)} c(e) \\
\geq \sum_{e \in B} \frac{f(e | A_{i-1})}{c(e)} c(e) \\
\geq f(B | A_{i-1}) \\
\geq f(B) - f(A_{i-1}).
\] (4.1)

Inequality (4.1) follows from the condition in the lemma, inequality (4.2) follows from the submodularity of $f$, and inequality (4.3) follows from the monotonicity of $f$. Thus, from the above we can conclude that:

\[
f(B) - f(A_{i}) \leq (f(B) - f(A_{i-1})) \left(1 - \frac{c(a_i)}{c(B)}\right).
\]

Hence,

\[
f(B) - f(A_{i}) \leq f(B) \prod_{j=1}^{i} \left(1 - \frac{c(a_j)}{c(B)}\right).
\]

Applying the inequality $1 - x \leq e^{-x}$ we get that:

\[
f(B) - f(A_{i}) \leq f(B) \cdot e^{-\frac{c(A_i)}{c(B)}}.
\]

Rearranging the terms and setting $i = k$ completes the proof.

A particular corollary of the above lemma is that if $A$ is the set of elements chosen by the greedy algorithm in the $k$ first iterations and $B$ is any subset such that no element from $B$ was dropped by the greedy algorithm in the first $k$ iterations, then all the condition of Lemma 15 are met and it can be applied to lower bound the value of $A$. 

41
4.3 Correct Proof for Approximation Factor of Modified Greedy

We note that the proof given by Khuller et al. [70] that the Modified Greedy algorithm (Algorithm 3) provides an approximation of \((1 - e^{-1/2})\) for Submodular Knapsack is flawed. The flaw originates from applying Lemma 13 incorrectly for a set for which the greedy algorithm (Algorithm 3) might have dropped elements. See the last statement in the proof of Theorem 3 in [70] (page 42, right column, line 10 from the bottom). To give a correct proof we incorporate a new insight to previous techniques for analyzing of the greedy algorithm (e.g. [70] and [87]). While previous approaches only showed lower bounds on the value of elements selected by the greedy before an element from the optimal solution has been dropped, we further look into the value attained by the algorithm after an element from the optimal solution has been dropped.

Let \(O\) be an optimal solution, and recall that \(T\) is the output of Algorithm 3.

Proof of Theorem 16. We can assume that \(f(e) \leq (1 - e^{-1/2})f(O)\) for every \(e \in U\), otherwise the value of \(e\), and hence of \(T\), is large enough and the theorem holds. Recall that by the definition of Algorithm 3 \(S\) is the output of the greedy algorithm, and let \(e^*\) be the most valuable element.

Consider two cases depending on how many elements of \(O\) are dropped by the greedy algorithm, i.e., \(|O \setminus S|\). If \(|O \setminus S| \leq 1\), then

\[
f(O) \leq f(O \cup S) \leq f(O \setminus S) + f(S) \leq f(e^*) + f(S).
\]

The above implies that either \(f(S) \geq \frac{1}{2}f(O)\) or \(f(e) \geq \frac{1}{2}f(O)\) concluding the proof for this case. Otherwise, \(|O \setminus S| \geq 2\). Denote by \(a\) and \(b\) the first and second elements of \(O\) the greedy algorithm drops, i.e. \(a\) is the considered element at the first time the condition in line 3 of Algorithm 3 is evaluated to false and \(b\) is the considered element the second time the same condition is evaluated to false. Denote by \(A\) the set of elements chosen by the algorithm just before dropping \(a\) and by \(B\) the set of elements chosen right after dropping \(a\) and before dropping \(b\). If \(f(A) \geq (1 - e^{-1/2})f(O)\) then the theorem holds. Otherwise, \(f(A) = (1 - e^{-1/2-\delta})f(O)\) where \(0 < \delta \leq 1/2\).

To prove the theorem we need to infer from the above bounds on both size and value. First, we focus on bounds on the size:

\[
c(A) \leq (\frac{1}{2} - \delta)c(O) \quad (4.4)
\]

\[
c(a) > (\frac{1}{2} + \delta)c(O) \quad (4.5)
\]

\[
c(b) \leq c(O \setminus a) \leq (\frac{1}{2} - \delta)c(O) \quad (4.6)
\]

Inequality (4.4) upper bounds the size of \(A\) and follows from Lemma 13 since if it does not hold then \(f(A) > (1 - e^{-(1/2-\delta)})f(O)\). Inequality (4.5) lower bound the size of \(a\) and follows from the fact that \(a\) was dropped by the greedy algorithm, i.e., \(c(A \cup \{a\}) > \beta \geq c(O)\). Inequality (4.6) is due to inequality (4.5), since \(c(O \setminus \{a\}) = c(O) - c(a)\) and \(\{b\} \subseteq O \setminus \{a\}\).

The above inequalities imply a lower bound on the size of \(B\):

\[
c(B) > (2\delta)c(O). \quad (4.7)
\]

Second, we focus on bounds on the value. We need to lower bound the value of \(O\) after \(a\) is dropped with respect to \(A\), i.e., \(f(O \setminus a|A)\):

\[
f(O \setminus a|A) \geq f(O) - f(a) - f(A) \geq (e^{-1/2} - 1 + e^{-(1/2-\delta)})f(O), \quad (4.8)
\]
where the inequality follows from the submodularity and monotonicity of $f$. Furthermore, we need to lower bound the value of the elements the greedy algorithm chooses after discarding $a$ and up to discarding $b$ with respect to $A$, i.e.,

\[ f(B | A) \geq (1 - e^{-(1/2-\delta)}) f(O \setminus a | A) \]
\[ \geq (1 - e^{-(1/2-\delta)}) (e^{-1/2} - 1 + e^{-(1/2-\delta)}) f(O). \]

Inequality (4.9) follows from applying Lemma 14 on the sets $B$ and $O \setminus a$, and considering the submodular function $f(X | A)$ for every $X \subseteq U$.

Finally, we lower bound the value of the output:

\[ f(T) \geq f(A \cup B) = f(A) + f(B | A). \]  

(4.10)

Substituting $f(A)$ with $(1 - e^{-(1/2-\delta)}) f(O)$ and lower bounding $f(B | A)$ using (4.9) results in the following:

\[ f(T) \geq (1 - e^{-(1/2-\delta)}) f(O) + (1 - e^{-(1/2-\delta)}) (e^{-1/2} - 1 + e^{-(1/2-\delta)}) f(O). \]  

(4.11)

One can verify that the expression in (4.11) is at least $(1 - e^{-1/2}) f(O)$ for $0 < \delta \leq 1/2$. 

\[ \square \]

### 4.4 The Modified Greedy Algorithm

We present a remarkably simple adaptation of the Modified Greedy algorithm of [70] (Algorithm 3) that retains the same running time of $O(n^2)$ for which we can prove a better approximation guarantee of 0.453647. Our algorithm returns the best between the greedy algorithm (Algorithm 2) and highest value pair of elements. We denote this algorithm by Modified Greedy and it appears as Algorithm 4. We prove that Algorithm 4 achieves the guarantee claimed in Theorem 17.

Algorithm 4: Modified Greedy$(U, f, c, \beta)$

1. $S \leftarrow$ greedy$(U, f, c, \beta)$
2. $T \leftarrow \arg \max \{S, \arg \max \{f(\{e_1, e_2\}) : e_1, e_2 \in U, c(\{e_1, e_2\}) \leq \beta\}\}$
3. return $T$

Proof of Theorem 14. Let $0 < \varepsilon \leq 1/6$ be an absolute constant to be determined later, and let $O$ be an optimal solution to the problem. If there is a pair of elements $e_1, e_2 \in U$ whose value is high enough, i.e., $f(\{e_1, e_2\}) \geq (1 - e^{-(1/2+\varepsilon)}) f(O)$, then the theorem follows. Thus, let us assume that $f(\{e_1, e_2\}) < (1 - e^{-(1/2+\varepsilon)}) f(O)$ for every pair of elements $e_1, e_2 \in U$.

Similarly to the proof of Theorem 16, we can assume that the greedy algorithm discards at least three elements from $O$, i.e., $|S \setminus O| \geq 3$. Otherwise, we are guaranteed that $f(T) \geq 1/2 \cdot f(O)$. Let $a$ be the first element in $O$ that was dropped by the algorithm, i.e., the first time the condition in line 3 was false, and let $A$ be the set of elements chosen by the algorithm just before dropping $a$. If $f(A) \geq (1 - e^{-(1/2+\varepsilon)}) f(O)$ the theorem holds. Otherwise, denote $f(A) = (1 - e^{-(1/2+\varepsilon-\delta)}) f(O)$ where $0 < \delta \leq 1/2 + \varepsilon$.

Lemma 13 implies that:

\[ c(A) \leq (1/2 + \varepsilon - \delta) c(O), \]  

(4.12)

43
otherwise \( f(A) > (1 - e^{-(1/2+\varepsilon-\delta)})f(O) \). Moreover, since \( a \) was dropped we confirm that:

\[
c(a) > (1/2 - \varepsilon + \delta)c(O), \tag{4.13}
\]

since \( c(A \cup \{a\}) > \beta \geq c(O) \). We say that an element \( e \) is big if \( c(e) \geq (1/4 + \varepsilon/2 - \delta/2)c(O) \), otherwise it is small. Note that \( a \) is big since \( \varepsilon \leq 1/6 \) and \( \delta > 0 \).

Note that \( O \setminus \{a\} \) contains at most one big element (if that is not the case then the size of two big elements is at least \((1/2 + \varepsilon - \delta)c(O)\)), in contradiction to (4.13). Thus, since the greedy algorithm dropped at least three elements from \( O \), it must be the case that at least one of them is small. Let \( b \) be the first such element, and let \( B \) be the set of elements chosen by the algorithm right after dropping \( a \) and just before dropping \( b \). Also, denote by \( C \) the subset of small elements in \( O \), i.e., \( C = \{e \in O : c(e) < (1/4 + \varepsilon/2 - \delta/2)c(O)\} \). We note that:

\[
c(C) \leq (1/2 + \varepsilon - \delta)c(O), \tag{4.14}
\]

since \( a \) is big. From the above inequalities we can derive a lower bound on the size of \( B \):

\[
c(B) \geq (1/4 - (3\varepsilon)/2 + (3\delta)/2)c(O). \tag{4.15}
\]

First, we wish to lower bound \( f(C) \). Since \( O \) contains at most two big elements, and the fact that the value of any pair of elements cannot exceed \((1 - e^{-(1/2+\varepsilon)})f(O)\), we can conclude that \( f(C) \geq e^{-(1/2+\varepsilon)} \) (follows from the submodularity of \( f \)).

Second, we aim to lower bound the marginal value of \( B \) with respect to \( A \). To that end we apply Lemma 15 on \( B \) and \( C \) and choose the submodular function to be \( f(X|A) \) for every \( X \subseteq U \). This gives us the following:

\[
f(B|A) \geq (1 - e^{-1/2+\varepsilon}) \left[ e^{-(1/2+\varepsilon)} - (1 - e^{-(1/2+\varepsilon-\delta)}) \right] f(O) \tag{4.16}
\]

Finally, we would like to lower bound \( f(T) \). Note that if \( f(A) \geq (1 - e^{-(1/2+\varepsilon)})f(O) \) we can lower bound \( f(T) \) by \((1 - e^{-(1/2+\varepsilon)})f(O) \). Otherwise,

\[
f(A \cup B) = f(A) + f(B|A) \\
\geq (1 - e^{-(1/2+\varepsilon)})f(O) + (1 - e^{-1/2+\varepsilon}) \left[ e^{-(1/2+\varepsilon)} - (1 - e^{-(1/2+\varepsilon-\delta)}) \right] f(O). \tag{4.17}
\]

Thus, for a given fixed \( \varepsilon > 0 \) we have:

\[
\frac{f(T)}{f(O)} \geq \min \left\{ \left(1 - e^{-(1/2+\varepsilon)}\right) \min \left\{ (1 - e^{-(1/2+\varepsilon-\delta)}) + (1 - e^{-1/2+\varepsilon}) \left[ e^{-(1/2+\varepsilon)} - (1 - e^{-(1/2+\varepsilon-\delta)}) \right] \right\} \right\} \tag{4.18}
\]

for \( 0 < \delta \leq 1/2 + \varepsilon \). Optimizing over \( 0 < \varepsilon \leq 1/6 \) we get that \( \varepsilon = 0.10449 \) and \( \delta = 0.388031 \) resulting in an approximation of 0.453647.

### 4.5 Amplification Algorithm

In this section we show a simple algorithm, which given an \( r \)-approximation \( A \) for **Submodular Knapsack**, \( r < 1/2 \) can be used to derive an \( r' \)-approximation, \( r < r' < 1/2 \), for **Submodular Knapsack** using a constant number of calls for \( A \) and \( \frac{3n^2 + n}{2} \) oracle queries.
Algorithm 5 receives an approximation algorithm $A$ for Submodular Knapsack, for a universe $U$, a monotone non-negative submodular function $f : 2^U \to \mathbb{R}$, a non-negative cost function $c : U \to \mathbb{R}$, and a budget $\beta$. The algorithm also gets two scalar parameters $\epsilon$ and $\rho$. These two parameters control the approximation ratio and complexity of the algorithm, and should be set according to the claims in this section to obtain the required approximation ratio and complexity.

Broadly speaking, the algorithm finds the pair of elements with maximal value $P_{\text{val}}$. It then divides all pairs of elements $\{a, b\} \subseteq U$ for which $f(\{a, b\}) \geq \rho \cdot \frac{f(P_{\text{val}})}{w_{\text{max}}}$ into buckets, where the subsets in each bucket have the same value to a factor of $(1 + \epsilon)$. The algorithm then extends the set of smallest cost in each bucket to a solution using the algorithm $A$. The algorithm returns the best between the solutions found using the buckets and $A$, the pair of elements with maximal profit, the single element with maximal profit, and the result of the greedy algorithm over the input instance.

Algorithm 5: Amplify($A, U, f, c, \beta, \epsilon, \rho$)

// Initialization
1 $P_{\text{val}} \leftarrow \arg\max_{\{a, b\} \subseteq U, c(\{a, b\)} \leq \beta f(\{a, b\})$
2 $w_{\text{max}} \leftarrow f(P_{\text{val}})$
3 $i_{\text{max}} \leftarrow \left\lfloor \log_{1+\epsilon} \frac{1}{\rho} \right\rfloor$

// Buckets
4 for $i \in \{0, \ldots, i_{\text{max}}\}$ do
5 $B_i = \{\{a, b\} \subseteq U | c(\{a, b\}) \leq \beta, \rho(1 + \epsilon)^i \leq \frac{f(\{a, b\})}{w_{\text{max}}} < \rho(1 + \epsilon)^{i+1}\}$
6 $P_i \leftarrow \arg\min_{\{a, b\} \in B_i} c(\{a, b\})$
7 $S_i \leftarrow P_i \cup A(U, f|_{P_i}, c, \beta - c(P_i))$
8 end
9 $G \leftarrow \text{Greedy}(U, f, c, \beta)$
10 $T \leftarrow \arg\max_{a \subseteq U, c(a)} f(\{a\})$
11 return $\arg\max_{S \in \{S_0, S_1, \ldots, S_{i_{\text{max}}} G, P_{\text{val}}, T\}} f(S)$

The following function, which already appeared on Theorem 18, will be useful throughout the analysis of the algorithm:

\[
B(\alpha) = 1 - e^{-\frac{\alpha}{2}}
\]

\[
A(\alpha) = \frac{1}{1 - e^{-\alpha}} - \frac{1}{B(\alpha)}
\]

\[
D(\alpha) = (1 - e^{-\alpha})/B(\alpha)
\]

Lemma 16. Given an $r$-approximation $A$ for Submodular Knapsack where $r < \frac{1}{2}$. For any $0 < \alpha \leq \ln 2$, input $U, f, c, \beta$ of Submodular Knapsack, and $\epsilon$ such that $0 < \epsilon \leq \frac{1 - 2r}{r}$, then executing Algorithm 5 with parameters $(A, U, f, c, \beta, \epsilon, A(\alpha))$ returns

\[
\min \left(1 - e^{-\alpha}, D(\alpha) + (1 - r) \left(\frac{2 + \epsilon}{1 + \epsilon} (1 - D(\alpha)) - 1\right)\right)
\]

approximation for the input instance.
Proof. Fix an optimal solution \( O \). If \(|O| \leq 2\) the algorithm returns an optimal solution and the Lemma holds. Therefore we can assume \(|O| \geq 3\). Denote by \( P_{\text{large}} \) the set of two elements in \( O \) with highest cost.

Let \( L \beta \) be the highest cost of element in \( O \). If \( L \leq 1 - \alpha \) then by Lemma 15 and the monotonicity of \( f \): \( f(G) \geq (1 - e^{-\alpha}) f(O) \) (recall that \( G \) is the result of greedy in line 9 of the algorithm and thus Lemma 15 can be applied on the appropriate prefix of \( G \). Therefore, the algorithm returns a solution of value at least \((1 - e^{-\alpha}) f(O)\) and the Lemma holds. Therefore we can assume \( L > 1 - \alpha \).

If \( O \setminus P_{\text{large}} \subseteq G \), then \( f(G) + f(P_{\text{val}}) \geq f(O \setminus P_{\text{large}}) + f(P_{\text{large}}) \geq f(O) \). Therefore \( f(G) \geq \frac{f(O)}{2} \) or \( f(P_{\text{val}}) \geq \frac{f(O)}{2} \) thus the Lemma holds (note that \( \alpha \leq \ln 2 \) and therefore \( 1 - e^{-\alpha} < \frac{1}{2} \)). Thus we can assume that \( O \setminus P_{\text{large}} \notin G \).

The following corollary states that if \( f(P_{\text{large}}) \) is fairly small with respect to \( f(O) \) (equivalently, \( f(O|P_{\text{large}}) \) is high) then the solution \( G \) from the greedy algorithm provides the required approximation ratio.

**Corollary 1.** If \( f(O|P_{\text{large}}) \geq D(\alpha) f(O) \) then \( f(G) \geq (1 - e^{-\alpha}) f(O) \).

**Proof.** Let \( \beta M \) be the third highest cost of element in \( O \). As the highest cost of element in \( O \) is \( \beta L \), using a simple argument we get, \( M \leq M(L) \) where \( M(L) = \min \{ \frac{1}{L} - \frac{1}{L} \} \). Since \( O \setminus P_{\text{large}} \notin G \), there must be an element from \( O \setminus P_{\text{large}} \) greedy in line 9 drops. Since all the elements in \( O \setminus P_{\text{large}} \) are of cost at most \( \beta M \) the knapsack must already have elements of cost \( \beta (1 - M) \) at the first time an element from \( O \setminus P_{\text{large}} \) is dropped. Also, \( c(O|P_{\text{large}}) \leq \beta (1 - L - M) \), therefore by Lemma 15 and the monotonicity of \( f \) we get that:

\[
f(G) \geq \left(1 - e^{-\frac{\beta (1-M)}{M(L)}}\right) f(O \setminus P_{\text{large}}) \geq \left(1 - e^{-\frac{1}{1-L-M}}\right) f(O \setminus P_{\text{large}}).
\]

We note that the term \( \frac{1-M}{1-L-M} = \left(1 - \frac{L}{1-M}\right)^{-1} \) is increasing as a function of \( M \) in the range \([0, M(L)]\) (note that \( 1 - L - M > 0 \) in the range). Therefore \( \frac{1-M}{1-L-M} \geq \frac{1}{1-L} \geq \frac{1}{1-\alpha} \). And conclude

\[
f(G) \geq \left(1 - e^{-\frac{1-M}{1-L-M}}\right) f(O \setminus P_{\text{large}}) \geq \left(1 - e^{-\frac{1}{1-L}}\right) f(O \setminus P_{\text{large}}) = B(\alpha) f(O \setminus P_{\text{large}})
\]

Now, recall that \( D(\alpha) = \frac{1-e^{-\alpha}}{B(\alpha)} \), \( f(O|P_{\text{large}}) \geq D(\alpha) f(O) \) by the condition of the lemma and \( f(O|P_{\text{large}}) \leq f(O \setminus P_{\text{large}}) \) as \( f \) is submodular. Using these observations and the last lower bound on \( f(G) \) we obtain

\[
f(G) \geq B(\alpha) f(O \setminus P_{\text{large}}) \geq B(\alpha) f(O|P_{\text{large}})
\]

\[
\geq B(\alpha) D(\alpha) f(O) = B(\alpha) \frac{1}{B(\alpha)} f(O) = (1 - e^{-\alpha}) f(O)
\]

\[\square\]

**Corollary 2.** If \( f(G) < (1 - e^{-\alpha}) f(O) \) and \( f(P_{\text{val}}) < (1 - e^{-\alpha}) f(O) \) then \( f(P_{\text{large}}) \geq A(\alpha) f(P_{\text{val}}) \)
Proof. As \( f(G) < (1 - e^{-\alpha}) \), the condition of corollary does not hold. Therefore \( f(O|P_{\text{large}}) < D(\alpha)f(O) \). Hence,

\[
f(O) = f(P_{\text{large}}) + f(O|P_{\text{large}}) < f(P_{\text{large}}) + D(\alpha)f(O)
\]

By rearranging the terms and using \( f(P_{\text{val}}) < (1 - e^{-\alpha})f(O) \) we get

\[
f(P_{\text{large}}) > (1 - D(\alpha))f(O) \]
\[
> \frac{1 - D(\alpha)}{1 - e^{-\alpha}} f(P_{\text{val}})
\]
\[
= \left( \frac{1}{1 - e^{-\alpha}} - \frac{D(\alpha)}{1 - e^{-\alpha}} \right) f(P_{\text{val}}) = A(\alpha)f(P_{\text{val}})
\]

The second inequality uses the observation that \( D(\alpha) \leq 1 \) for \( 0 \leq \alpha \leq 1 \) which can be easily verified. The last equality follows from the definitions of \( D(\alpha) \) and \( A(\alpha) \).

\[ \square \]

**Corollary 3.** If \( f(G) < (1 - e^{-\alpha})f(O) \) and \( f(P_{\text{val}}) < (1 - e^{-\alpha})f(O) \) then there is \( i \in \{0, \ldots, i_{\text{max}}\} \) such that

\[
\frac{f(S_i)}{f(O)} \geq D(\alpha) + (1 - r) \left( \frac{2 + \epsilon}{1 + \epsilon} D(\alpha) - 1 \right)
\]

Proof. As the conditions of corollary hold, we have \( f(P_{\text{large}}) \geq A(\alpha)f(P_{\text{val}}) \). Let \( i \) be the minimal integer \( i \) such that \( \frac{f(P_{\text{large}})}{f(P_{\text{val}})} < A(\alpha)(1 + \epsilon)^{i+1} \), as \( f(P_{\text{large}}) \geq A(\alpha)f(P_{\text{val}}) \) we get \( i \geq 0 \). Recall that

\[
i_{\text{max}} = \left\lfloor \log_{1+\epsilon} \frac{1}{\rho} \right\rfloor = \left\lfloor \log_{1+\epsilon} \frac{1}{A(\alpha)} \right\rfloor \text{ (as the algorithm is used with } \rho = A(\alpha))
\]

Therefore,

\[
A(\alpha)(1 + \epsilon)^{i_{\text{max}}+1} > A(\alpha) \frac{1}{A(\alpha)} = 1 \geq \frac{f(P_{\text{large}})}{f(P_{\text{val}})}
\]

Thus \( i \leq i_{\text{max}} \). Also \( c(P_{\text{large}}) \leq c(O) \leq \beta \), and we can conclude that \( P_{\text{large}} \in B_i \) (note that \( w_{\text{max}} = f(P_{\text{val}}) \)).

From the definition of \( P_i \) we get \( c(P_i) \leq c(P_{\text{large}}) \) and \( f(P_i) \geq \frac{1}{1+r} f(P_{\text{large}}) \). Let \( Q_i = A(U, f_{P_i}, c, \beta - c(P_i)) \). As \( c(P_i) \leq c(P_{\text{large}}) \) we get that \( O \setminus P_{\text{large}} \) is a feasible solution for the problem instance given to \( A \). Hence, as \( A \) is a \( r \)-approximation we get

\[
f(Q_i|P_i) \geq rf(O|P_{\text{large}}|P_i)
\]

Therefore,

\[
f(S_i) \geq f(P_i) + f(Q_i|P_i) \]
\[
\geq f(P_i) + rf(O|P_{\text{large}}|P_i) \]
\[
\geq f(P_i) + rf(O) - f(P_{\text{large}}) - f(P_i) \]
\[
= (1 - r)f(P_i) + rf(O) - f(P_{\text{large}}) \]
\[
= \frac{1 - r}{1 + \epsilon} f(P_{\text{large}}) + rf(O) \]
\[
= \left( \frac{2 + \epsilon}{1 + \epsilon} (1 - r) - 1 \right) f(P_{\text{large}}) + rf(O)
\]

47
As \( \epsilon \leq \frac{1-2r}{r} \) one can deduce that \( \left( \frac{2}{1+r} \right) (1-r) - 1 \geq 0 \). Also, since \( f(G) < (1 - e^{-\nu}) f(O) \) by corollary 10 we get \( f(O|P_{\text{large}}) < D(\alpha) f(O) \), by using \( f(O|P_{\text{large}}) = f(O) - f(P_{\text{large}}) \) and rearranging the terms we get \( f(P_{\text{large}}) > (1 - D(\alpha)) f(O) \). Therefore,

\[
f(S_i) \geq \left( \frac{2 + \epsilon}{1 + \epsilon} \right) (1-r) - 1 \right) f(P_{\text{large}}) + rf(O)
\geq \left( \frac{2 + \epsilon}{1 + \epsilon} \right) (1-r) - 1 \right) (1 - D(\alpha)) f(O) + rf(O)
= f(O) \left( D(\alpha) + (1-r) \left( \frac{2 + \epsilon}{1 + \epsilon} (1 - D(\alpha)) - 1 \right) \right)
\]
and the corollary immediately follows.

The Lemma follows immediately from corollary 10 as it states that either \( G, P_{\text{cal}} \) or \( S_i \) for some \( i \) would provide the required approximation ratio.

**Lemma 17.** Algorithm 3 uses \( \left\lfloor \log_{1+r} \frac{1}{\rho} \right\rfloor + 1 \) invocation to \( A \) and up to \( \frac{3}{2}n^2 + n \) additional oracle queries.

**Proof.** The number of invocation to \( A \) immediately follows from the algorithm. Beside the invocation to \( A \) the algorithm runs the greedy procedure which uses up to \( n^2 \) queries. The queries for the initialization and buckets phases only considers sets of size 2, and therefore can be implemented by up to \( n(n-1)/2 \leq n^2/2 \) queries. The execution of line 10 would take another \( n \) queries.

**Proof of Theorem 19.** Let

\[
\alpha^* = \arg \max_{0 < \alpha \leq \ln 2} \left\{ \min \left\{ 1 - e^{-\alpha}, D(\alpha) + (1-r) \left( \frac{2 + \nu(\alpha)}{1 + \nu(\alpha)} (1 - D(\alpha)) - 1 \right) \right\} \right\}
\]

We obtain the described approximation ratio when running the algorithm with \( \epsilon = \nu(\alpha^*) \) and \( \rho = A(\alpha^*) \). By the conditions of the theorem,

\[
k \geq - \frac{\log A(\ln 2)}{\log(1/r - 1)} + 1 \geq - \frac{\log A(\alpha^*)}{\log(1/r - 1)} + 1
\]

Where the last inequality uses the fact that \( A \) is monotonically decreasing. Thus,

\[
\epsilon^* = 2^{-\frac{\log A(A^*)}{k-1}} - 1 \leq 2^{-\left( \frac{\log A(A^*)}{\log_2(1/r - 1)} \right)} - 1 = 2^{\log_2(1/r - 1)} - 1 = 1/r - 2 = \frac{1 - 2r}{r}
\]

therefore the conditions of Lemma 16 apply, and the approximation ratio follows.

By lemma 17, the number of invocations for \( A \) is

\[
\left\lfloor \log_{1+\epsilon^*} \frac{1}{A(\alpha^*)} \right\rfloor + 1 \leq - \log_{1+\epsilon^*}(A(\alpha^*)) + 1 = - \frac{\log A(\alpha^*)}{\log(1 + \epsilon^*)} + 1
= - \frac{\log A(\alpha^*)}{\log(2^{-\frac{\log A(\alpha^*)}{k-1}})} + 1 = - \frac{\log A(\alpha^*)}{\frac{1}{\log_2(A(\alpha^*))}} + 1 + 1 = k
\]

and the number of addition oracle queries is \( 3n^2/2 + n \) as required.
Chapter 5

Maximum Carpool Matching

5.1 Introduction

As traveling costs become higher and parking becomes sparse it is only natural to share rides or to carpool. Originally, carpooling was an arrangement among a group of people by which they take turns driving the others to and from a designated location. However, taking turns is not essential, instead passengers can share the cost of the ride with the driver. Carpooling has social advantages other than reducing the costs: it reduces fuel consumption and road congestion and frees parking space. While in the past carpooling was usually a fixed arrangement between friends or neighbors, the emergence of social networks has made carpooling more dynamic and widespread. These days applications like Zimride [4], BlaBlaCar [1], Moovit [2] and even Waze [3] are matching passengers to drivers.

The matching process of passengers to drivers entails more than matching the route. Passenger satisfaction also need to be taken into account. Given several riding options (including taking their own car), passengers have preferences. For example, a passenger may prefer to ride with a co-worker or a friend. A passenger may have an opinion on a driver that she rode with in the past. She may prefer a non-smoker, someone who shares her taste in music, or someone who is recommended by others. Moreover, the matching process may take into account driver preferences. For instance, we would like to minimize the extra distance that a driver has to take. Preferences may also be computed using past information. Knapen et al. [72] described an automatic service to match commuting trips. Users of the service register their personal profile and a set of periodically recurring trips, and the service advises registered candidates on how to combine their commuting trips by carpooling. The service estimates the probability that a person $a$ traveling in person’s $b$ car will be satisfied by the trip. This is done based on personal information and feedback from users on past rides.

In this paper we assume that potential passenger-driver satisfactions are given as input and the goal is to compute an assignment of passengers to drivers so as to maximize the global satisfaction. More formally, we are given a directed graph $G = (V, A)$, where each vertex $v \in V$ corresponds to a user of the service, and an arc $(u, v)$ exists if the user corresponding to vertex $u$ is willing to commute with the user corresponding to vertex $v$. We are given a capacity function $c : V \to \mathbb{N}$ which bounds the number of passengers each user can drive if she is selected as a driver. A non-negative weight function $w : A \to \mathbb{R}^+$ is
used to model the amount of satisfaction $w(u, v)$ of assigning $u$ to $v$. If $(u, v) \in A$ implies that $(v, u) \in A$ and $w(u, v) = w(v, u)$, the instance is undirected. If $w(v, u) = 1$, for every $(v, u) \in A$, then the instance is unweighted. If $c(v) = \deg(v)$, for every $v$, then the instance is uncapacitated.

Given a directed graph $G$ and a subset $M \subseteq A$, define $\deg^M_{\text{in}}(v) \triangleq |\{u : (u, v) \in M\}|$ and $\deg^M_{\text{out}}(v) \triangleq |\{u : (v, u) \in M\}|$. A feasible carpool matching is a subset of arcs, $M \subseteq A$, such that every $v \in V$ satisfies: (i) $\deg^M_{\text{in}}(v) \cdot \deg^M_{\text{out}}(v) = 0$, (ii) $\deg^M_{\text{in}}(v) \leq c(v)$, and (iii) $\deg^M_{\text{out}}(v) \leq 1$. A feasible carpool matching $M$ partitions $V$ as follows:

$$P_M \triangleq \{v : \deg^M_{\text{out}}(v) = 1\}$$

$$D_M \triangleq \{v : \deg^M_{\text{in}}(v) \geq 1\}$$

$$Z_M \triangleq \{v : \deg^M_{\text{in}}(v) = \deg^M_{\text{out}}(v) = 0\}$$

where $P_M$ is the set of passengers, $D_M$ is the set of active drivers, and $Z_M$ is the set of solo drivers. In the Maximum Carpool Matching problem the goal is to find a matching $M$ of maximum total weight, namely to maximize $w(M) \triangleq \sum_{(v, u) \in M} w(v, u)$. In other words, the Maximum Carpool Matching problem is about finding a set of (directed toward the center) vertex disjoint stars that maximizes the total weight of the arcs. Figure 5.1 contains an example of a Maximum Carpool Matching instance. Note that in the unweighted case the goal is to find a carpool matching $M$ that maximizes $|P_M|$. Moreover, observe that if $G$ is undirected, $D_M \cup Z_M$ is a dominating set. Hence, in this case, an optimal carpool matching induces an optimal dominating set and vice versa. Since Minimum Dominating Set is NP-hard, it follows that Maximum Carpool Matching is NP-hard even if the instance is undirected, unweighted, and uncapacitated.

We also consider an extension of Maximum Carpool Matching, called Maximum
Related work. Agatz et al. [5] outlined the optimization challenges that arise when developing technology to support ride-sharing and survey the related operations research models in the academic literature. Hartman et al. [58] designed several heuristic algorithms for the Maximum Carpool Matching problem and compared their performance on real data. Other heuristic algorithms were developed by Knappen et al. [73]. Hartman [57] proved that the Maximum Carpool Matching problem is NP-hard even in the case where the weight function is binary and \( c(v) \leq 2 \) for every \( v \in V \). In addition, Hartman presented a natural integer linear program and showed that if the set of drivers is known, then an optimal assignment of passengers to drivers can be found in polynomial time using a reduction to Network Flow (see also [83].) Kutiel [83] presented a \( \frac{1}{7} \)-approximation algorithm for Maximum Carpool Matching that is based on a Minimum Cost Flow computation and a local search \( \frac{1}{2} \)-approximation algorithm for the unweighted variant of Maximum Carpool Matching. The latter starts with an empty matching and tries to improve the matching by turning a single passenger into a driver.

Nguyen et al. [96] considered the Spanning Star Forest problem. A star forest is a graph consisting of vertex-disjoint star graphs. In the Spanning Star Forest problem, we are given an undirected graph \( G \), and the goal is to find a spanning subgraph which is a star forest that maximizes the weight of edges that are covered by the star forest. Notice that this problem is equivalent to Maximum Carpool Matching on undirected and uncapacitated instances. We also note that if all weights leaving a vertex are the same, then the instance is referred to as vertex-weighted. Nguyen et al. [96] provided a PTAS for unweighted planner graphs and a polynomial-time \( \frac{2}{3} \)-approximation algorithm for unweighted graphs. They gave an exact optimization algorithm for weighted trees, and used it on a maximum spanning tree of the input graph to obtain a \( \frac{1}{2} \)-approximation algorithm for weighted graphs. They also showed that it is NP-hard to approximate unweighted Spanning Star Forest within a ratio of \( \frac{259}{260} + \varepsilon \), for any \( \varepsilon > 0 \). Chen et al. [27] improved the approximation ratio for unweighted graphs from \( \frac{3}{5} \) to 0.71 and gave a 0.64-approximation algorithm for vertex weighted graphs. They also showed that the edge- and vertex-weighted problem cannot be approximated to within a factor of \( \frac{19}{18} + \varepsilon \) and \( \frac{31}{32} + \varepsilon \), resp., for any \( \varepsilon > 0 \), assuming that \( P \neq \text{NP} \). Chakrabarty and Goel [24] improved the lower bounds to \( \frac{10}{11} + \varepsilon \) and \( \frac{12}{13} \).

Athanassopoulos et al. [11] improved the ratio for the unweighted case to \( \frac{193}{200} \approx 0.804 \). They considered a natural family of local search algorithms for Spanning Star Forest. Such an algorithm starts with the solution where all vertices are star centers. Then, it repeatedly tries to turn \( t \leq k \) from leaves to centers and \( t + 1 \) centers to leaves. A change is made if it results in a feasible solution, namely if each leave is adjacent to at least one center. The algorithm terminates when such changes are no longer possible. Athanassopoulos et al. [11] showed that, for any \( k \) and \( \varepsilon \in (0, \frac{2}{7(k+1)}) \), there exists an instance \( G \) and a local optima whose size is smaller than \( (\frac{1}{2} + \varepsilon) \text{OPT} \), where \( \text{OPT} \) is the size of the optimal spanning star forest. We note that, for a given \( k \), the construction of the above result requires that the maximum degree of \( G \) is at least \( 2(k+2) \). Hence, this result does not hold in graphs
with maximum degree $\Delta$.

Arkin et al. [10] considered the **Maximum Capacitated Star Packing** problem. In this problem the input consists of a complete undirected graph with non-negative edge weights and a capacity vector $c = \{c_1, \ldots, c_p\}$, where $\sum_{i=1}^{p} c_i = |V| - p$. The goal is to find a set of vertex-disjoint stars in $G$ of size $c_1, \ldots, c_p$ of maximum total weight. Arkin et al. [10] provided a local search algorithm whose approximation ratio is $\frac{1}{2}$, and a matching-based $\frac{1}{2}$-approximation algorithm for the case where edge weights satisfy the triangle inequality.

Bar-Noy et al. [16] considered the **Minimum 2-Path Partition** problem. In this problem the input is a complete graph on $3k$ vertices with non-negative edge weights, and the goal is to partition the graph into disjoint paths of length $2$. This problem is the special case of the undirected carpool matching where $c(v) = 2$, for every $v \in V$. They presented two approximation algorithms, one for the weighted case whose ratio is $0.5833$, and another for the unweighted case whose ratio is $\frac{3}{5}$.

Another related problem is **$k$-SET PACKING**, where one is given a collection of weighted sets, each containing at most $k$ elements, and the goal is to find a maximum weight sub-collection of disjoint sets. Chandra and Halldórsson [25] presented a $\frac{3}{2k+1}$-approximation algorithm for this problem. **Maximum Carpool Matching** can be seen as a special case of $k$-SET PACKING with $k = c_{\text{max}} + 1$. Consider a subset of vertices $U$ of size at most $k$. Observe that each subset of vertices has an optimal internal assignment of passenger to drivers. Let the weight of this assignment be the profit of $U$, denote by $p(U)$. If $k = O(1)$, $p(U)$ can be computed for every $U$ of size at most $k$ in polynomial time. The outcome is a $k$-SET PACKING instance. This leads to a $\frac{1}{2(\text{max } c + 2)}$-approximation algorithm when $c_{\text{max}} = O(1)$.

**Our contribution.** Section 5.2 contains approximation algorithms for **Maximum Carpool Matching**. First, in Section 5.2.1 we show that **Maximum Carpool Matching** can be formulated as an unconstrained submodular maximization problem, thus it has a $\frac{1}{2}$-approximation algorithm due to [21, 20]. We present a local search algorithm for **Maximum Carpool Matching** which repeatedly checks whether the current carpool matching can be improved by means of a star centered at a vertex, and it terminates when such a step is not possible. The approximation ratio of this algorithm is $\frac{1}{2}$ if weights are polynomially bounded, and its ratio is $\frac{1}{2} - \varepsilon$ in general.

In Section 5.3 we consider **Maximum Carpool Matching** with bounded maximum capacity. In Section 5.3.1 we show that **Maximum Carpool Matching** is APX-hard even for undirected and unweighted instances with $\Delta \leq b$, for any $b \geq 3$. In Section 5.3.2 we provide another local search algorithm, whose approximation ratio is $\frac{1}{2} + \frac{1}{2\text{max}} - \varepsilon$, for any $\varepsilon > 0$, for unweighted **Maximum Carpool Matching**, where $c_{\text{max}} \triangleq \max_{v \in V} c(v)$. Given a parameter $k$, our algorithm starts with the empty carpool matching. Then, it repeatedly tries to find a better matching by replacing $t \leq k$ arcs in the current solution by $t + 1$ arcs that are not in the solution. We show that our analysis is tight. We also note that our algorithm falls within the local search family defined in [11]. However, on undirected and uncapacitated instances we have that $c_{\text{max}} = \Delta$, and as mentioned above the result from [11] does not hold in bounded degree graphs.

Finally, Section 5.4 discusses **Maximum Group Carpool Matching**. We show that the unconstrained submodular maximization formulation for **Maximum Carpool Matching** does not work for **Maximum Group Carpool Matching**. We show, however, that this problem still admits a $(\frac{1}{2} - \varepsilon)$-approximation algorithm by extending our first local search algorithm. In addition, we show that the second local search algorithm generalizes to
5.2 Approximation Algorithms

We present two algorithms for Maximum Carpool Matching: a $\frac{1}{2}$-approximation algorithm that is based on formulating the problem as an unconstrained submodular maximization problem and a local search $(\frac{1}{2} - \varepsilon)$-approximation algorithm. While the latter does not improve upon the former, it will be shown in Section 5.4 that it can be generalized to Maximum Group Carpool Matching without decreasing the approximation ratio.

5.2.1 Submodular Maximization

In this section we show that the Maximum Carpool Matching problem can be formulated as an unconstrained submodular maximization problem, and thus it has a $\frac{1}{2}$-approximation algorithm due to Buchbinder et al. [21, 20].

Given a Maximum Carpool Matching instance $(G = (V,A), c, w)$, consider a subset $S \subseteq V$. Let $M(S)$ be a maximum weight carpool matching satisfying $D_M(S) \subseteq S \subseteq V \setminus S$, namely $M(S)$ is the best carpool matching whose drivers belong to $S$ and whose passengers belong to $V \setminus S$. In other words, $M(S)$ is the maximum weight carpool matching that is a subset of $A \cap (V \setminus S) \times S$. Given $S$, the carpool matching $M(S)$ can be computed in polynomial time by computing a maximum $b$-matching in the bipartite graph $B = (V \setminus S, S, A \cap (V \setminus S) \times S)$ which can be done using an algorithm for Minimum Cost Flow as shown in [83].

Consider the function $w : 2^V \rightarrow \mathbb{R}$, where $w(S) = \sum_{e \in M(S)} w(e)$. Observe that $w(\emptyset) = \emptyset(V) = 0$, and that $w$ is not monotone. In the next lemma we prove that $w$ is a submodular set function. Recall that a function $f$ is submodular if $f(S) + f(T) \geq f(S \cup T) + f(S \cap T)$ for every two sets $S$ and $T$ in the domain of $f$.

**Lemma 18.** $w$ is submodular.

**Proof.** Consider any two subsets $S, T \subseteq V$. We show that $w(S) + w(T) \geq w(S \cup T) + w(S \cap T)$. Let $M(S \cup T)$ and $M(S \cap T)$ be optimal carpool matchings with respect to $S \cup T$ and $S \cap T$. To prove the lemma we construct two feasible carpool matchings $M_S$ and $M_T$ such that $M_S \subseteq (V \setminus S) \times S$, $M_T \subseteq (V \setminus T) \times T$, and $M_S \cup M_T = M(S \cup T) \cup M(S \cap T)$. The lemma follows, since $w(S) \geq w(M_S)$ and $w(T) \geq w(M_T)$.

First, add all the edges in $M(S \cup T)$ entering $S \setminus T$ to $M_S$. Similarly, add all the edges in $M(S \cup T)$ entering $T \setminus S$ to $M_T$. Observe that $\deg_{in}^M(v) = \deg_{in}^M(v) \leq c(v)$, for every $v \in S \setminus T$ and that $\deg_{in}^M(v) = \deg_{in}^M(v) \leq c(v)$, for every $v \in T \setminus S$. Next, add the edges in $M(S \cap T)$ leaving $T \setminus S$ to $S$ and add the edges in $M(S \cap T)$ leaving $S \setminus T$ to $T$. It remains to distribute the edges leaving $V \setminus (S \cup T)$ and entering $S \cap T$ in both $M(S \cup T)$ and $M(S \cap T)$. Note that there may exist edges $(v,u)$, where $v \notin S \cup T$, and $u \in S \cap T$ such that $(v, u) \in M(S \cup T)$ and $M(S \cap T)$. We refer to these edges as duplicate edges. We add all edges leaving $V \setminus (S \cup T)$ and entering $S \cap T$ in $M(S \cap T)$ to $M_S$. Notice that this is possible, since after this addition we have that $\deg_{in}^M(v) \leq \deg_{in}^M(v) \leq c(v)$, for every vertex $v \in S \cap T$. Then we add all duplicate edges in $M(S \cup T)$ to $M_T$. The remaining edges are distributed between $M_S$ and $M_T$ without violating capacities. This can be done, since $\deg_{in}^M(v) + \deg_{in}^M(v) \leq 2c(v)$, for every $v \in S \cap T$. Figure 5.2 contains a graphical representation of the above edge distribution.

53
Figure 5.2: Given the two solutions on the left, (a) and (c), we can construct the two solutions on the right side, (b) and (d), with equal total weight. The edges of type \( \rightarrow \), \( \rightarrow \), \( \rightarrow \), \( \rightarrow \) from the left side are “naturally” distributed over the two solutions on the right side and the edges of type \( \rightleftarrows \) and \( \rightleftarrows \) can be distributed between the two solutions without violating the capacity constraints.

Buchbinder et al. [21, 20] presented a general \( \frac{1}{2} \)-approximation algorithm for unconstrained submodular maximization, thus we have the following theorem.

**Theorem 19.** There exists a polynomial time \( \frac{1}{2} \)-approximation algorithm for **Maximum Carpool Matching**.

### 5.2.2 A Star Improvement Algorithm

In this section we give a local search \((\frac{1}{2} - \varepsilon)\)-approximation algorithm for **Maximum Carpool Matching**. This algorithm repeatedly checks whether the current carpool matching \( M \) can be improved by means of a star centered at a vertex \( v \). The profit from this star is the total weight of the arcs in the star, and the cost is the total weight of lost arcs (e.g., arcs from passengers to drivers that became passengers of \( v \)). If the profit is larger than the cost, then an improvement step is performed. The algorithm terminates when such a step is not possible. We remind the reader that this algorithm will be extended to **Maximum Group Carpool Matching** in Section 5.4.

We need a few definitions before presenting our algorithm. Given a directed graph \( G = (V, A) \), define \( N^{\text{in}} \doteq \{ u : (u, v) \in A \} \) and \( N^{\text{out}} \doteq \{ u : (v, u) \in A \} \). Let \( M \) be a feasible carpool matching. The weight \( w_M(v) \) of a vertex \( v \) with respect to \( M \) is the sum of the weights of the arcs in \( M \) that are incident on \( v \), namely

\[
w_M(v) \doteq w(M \cap N^{\text{in}}) + w(M \cap N^{\text{out}}) = \sum_{(u,v) \in M} w(u,v) + \sum_{(v,u) \in M} w(v,u).
\]

For a subset of vertices \( U \subseteq V \) we define \( w_M(U) \doteq \sum_{v \in U} w_M(v) \).
Figure 5.3: In this example $M$ is the set of the blue, dashed arcs. In this case $w_M(2) = 7$, $w_M(5) = 2$, and $w_M(6) = 0$. Also, $\delta_M(2, 3) = 1$ and $\delta_M(6, 3) = 2$. The set $\{(2, 3), (6, 3)\}$ is an improvement to vertex 3 and $\Gamma(2, 3) = \{(1, 2), (4, 2), (3, 5), (6, 3)\}$.

We now argue that, with respect to any carpool matching $M$, the total weight of all the vertices is equal to twice the weight of the matching.

**Observation 3.** $w_M(V) = 2w(M)$.

**Proof.** $\sum_{v \in V} w_M(v) = \sum_{v \in V} \sum_{(u, v) \in M} w(u, v) + \sum_{v \in V} \sum_{(v, u) \in M} w(v, u) = 2 \sum_{e \in E} w(e)$. \hfill \Box$

Denote by $\delta(u, v)$ the difference between the weight of the arc and the weight of its source vertex, that is:

$\delta_M(u, v) \triangleq w(u, v) - w_M(u)$.

For a subset $S \subseteq A$ of arcs define $\delta(S) \triangleq \sum_{(u, v) \in S} \delta(u, v)$.

A subset $S_v$ of arcs entering a vertex $v$, whose size is not greater than the capacity of $v$, is called an improvement to vertex $v$ if $\delta(S_v)$ is greater than the value of $v$. More formally,

**Definition 1.** A subset $S_v \subseteq A \cap (V \times \{v\})$ is an improvement with respect to a carpool matching $M$, if (i) $|S_v| \leq c(v)$, and (ii) $\delta_M(S_v) > w_M(v)$. Furthermore, if there exists an improvement for a vertex $v$, we say that vertex $v$ can be improved.

Given an arc $(u, v) \in A$, let $\Gamma(u, v)$ be the set of arcs that incident $(u, v)$, namely $\Gamma(u, v) \triangleq N^{in}(u) \times \{u\} \cup \{(v) \times N^{out}(v)\}$. If $S$ is a set of arcs, then $\Gamma(S) \triangleq \bigcup_{(u, v) \in S} \Gamma(u, v)$. Figure 5.3 depicts all the above definitions.

We are now ready to describe our local search algorithm, which is called **StarImprove** (Algorithm 4). It starts with an empty carpool matching $M$, and in every iteration it looks for a vertex that can be improved. If there exists such a vertex $v$, then the algorithm removes the arcs that are incident on it from $M$, and adds the arcs in $S_v$. The algorithm terminates when no vertex can be improved. Figure 5.4 depicts an improvement step.

We proceed to bound the approximation ratio of the algorithm, assuming termination.

For a vertex $v$ and a set $S$ of edges entering $v$, let $N^{in}_S(v) = \{u : (u, v) \in S\}$ be the set in-neighbors corresponding to $S$.

**Lemma 19.** Let $M$ be a matching computed by **StarImprove**. Let $v$ be a vertex with no improvement, and let $S \subseteq N^{in}_M(v)$, such that $|S| \leq c(v)$, then $w(S) \leq w_M(v) + w_M(N^{in}_S(v))$.

**Proof.** If no improvement exists, then we have that

$w(S) - w_M(N^{in}_S(v)) = \sum_{(u, v) \in S} (w(u, v) - w_M(u)) = \delta_M(S) \leq w_M(v)$,

as required. \hfill \Box
Algorithm 6: StarImprove(G, c)

1. \[ M \leftarrow \emptyset \]
2. repeat
3. \[ \text{done} \leftarrow \text{TRUE} \]
4. \[ \text{for } v \in V \text{ do} \]
5. \[ \text{if there exists an improvement } S_v \text{ then} \]
6. \[ M \leftarrow M \setminus \Gamma(S_v) \cup S_v \]
7. \[ \text{done} \leftarrow \text{FALSE} \]
8. \[ \text{end} \]
9. \[ \text{end} \]
10. until \text{done};

(a) A matching that can be improved.

(b) The matching after improving vertex 3.

Figure 5.4: An improvement example.
To bound the approximation ratio of the algorithm, we use a charging scheme argument.

**Lemma 20.** If StarImprove terminates, then the computed solution is $\frac{1}{2}$-approximate.

**Proof.** Let $M$ be the matching produced by the algorithm, and let $M^*$ be an optimal matching. We load every vertex $v$ with an amount of money equal to $w_M(v)$, and then we show that this is enough to pay for every arc in the optimal matching. Due to Observation 3, the total amount of money that we use is exactly twice the weight of $M$.

Consider a driver $v \in D_M$, and let $S = (V \times \{v\}) \cap M^*$. By Lemma 19, we know that $w(S) \leq w_M(v) + w_M(N^S(v))$, thus we can pay for $S$, using the money on $v$ and on $N^S(v)$. Clearly, these vertices will not be charged again.

We show that our analysis is tight using Figure 5.5.

It remains to consider the running time of the algorithm.

**Theorem 20.** Algorithm StarImprove is a $\frac{1}{2}$-approximation algorithm for Maximum Carpool Matching, if edge weights are integral and polynomially bounded.

**Proof.** First, observe that determining if a vertex $v$ can be improved can be done efficiently by considering the incoming arcs to $v$ in a non-increasing order of their $\delta_M$, and only ones with positive values. A vertex $v$ can be improved, then, if the $\delta$s of the first $c(v)$ (or less) arcs sum up to more than $w_M(v)$. It follows that the running time of an iteration of the for-loop is polynomial. Since the edge weights are integral and polynomially bounded, the weight of an optimal carpool matching is polynomially bounded. The algorithm runs in polynomial time, because in each iteration the algorithm improves the weight of the matching by at least one or otherwise it terminates.

It remains to consider the case, where weight are not polynomially bounded. We prove that one can use standard scaling and rounding to ensure a polynomial running time in the cost of a slight degradation of the approximation ratio.

**Theorem 21.** There exists a $(\frac{1}{2} - \varepsilon)$-approximation algorithm for Maximum Carpool Matching, for every $\varepsilon \in (0, \frac{1}{2})$.

**Proof.** Define the weight function $w'(e) = \left\lfloor \frac{w(e)}{w_{\max}} \cdot \frac{m}{2\varepsilon} \right\rfloor$, where $w_{\max} = \max_{e \in A} w(v)$. Let $M^*$ and $M'$ be an optimal carpool matchings with respect to $w$ and $w'$, resp. We have that

$$w'(M^*) = \sum_{e \in M^*} \left\lfloor \frac{w(e)}{w_{\max}} \cdot \frac{m}{2\varepsilon} \right\rfloor > \sum_{e \in M^*} \left( \frac{w(e)}{w_{\max}} \cdot \frac{m}{2\varepsilon} \right) - m = \frac{m}{2\varepsilon w_{\max}} \cdot w(M^*) - m.$$

Since $w(M^*) \geq w_{\max}$, we have that

$$w'(M^*) > \frac{m}{2\varepsilon w_{\max}} \cdot (w(M^*) - 2\varepsilon w_{\max}) \geq \frac{m}{2\varepsilon w_{\max}} \cdot (1 - 2\varepsilon)w(M^*).$$
By Theorem 20, our local improvement algorithm computes a \(1/2\)-approximate carpool matching \(M\) on \((G, c, w')\), and this matching satisfies \(w(M) \geq w'(M')/2\). Furthermore, since \(w'(M') \geq w'(M^*)\), it follows that \(w(M) \geq w'(M^*)/2\). Therefore
\[
w(M) \geq \frac{2\varepsilon w_{\max}}{m} w(M) \geq \frac{1}{2} \frac{2\varepsilon w_{\max}}{m} w'(M^*) > \frac{1 - 2\varepsilon}{2} w(M^*) ,
\]
as required.

### 5.3 Constant Maximum Capacity

In this section we study the Maximum Carpool Matching problem when the maximum capacity is constant, i.e., when \(c_{\max} = O(1)\). We show that this variant of the problem is APX-hard even for unweighted and undirected instances. We also describe and analyze a local search algorithm for the unweighted variant of the problem, and show that the algorithm achieves a \(1/2 + 1/(2c_{\max}) - \varepsilon\) approximation ratio, for any \(\varepsilon > 0\).

#### 5.3.1 Hardness

As we mentioned earlier, Spanning Star Forest has a lower bound of \(10/11 + \varepsilon\) for any \(\varepsilon > 0\), unless \(P = NP\) [24], and this bound applies to Maximum Carpool Matching. The result, however, does not hold for the case where \(\Delta = O(1)\) (and \(c_{\max} = O(1)\)). In this section we show that the problem remains APX-hard even in this case.

Formally, the (unweighted) Minimum Dominating Set problem is defined as follows. The input is an undirected graph \(G = (V, E)\), and a feasible solution, or a dominating set, is a subset \(D \subseteq V\) that dominates \(V\) namely such that \(D \cup \bigcup_{v \in D} N(v) = V\), where \(N(v)\) is the neighborhood of \(v\). The goal is to find a minimum cardinality dominating set. Minimum Dominating Set-b is the special case of Minimum Dominating Set in which the maximum degree of a vertex in the input graph \(G\) is bounded by \(b\). The problem was shown to be APX-hard, for \(b \geq 3\), by Papadimitriou and Yannakakis [97].

We now consider the unweighted and undirected special case of the Maximum Carpool Matching problem. In this case, the input consists of an undirected graph \(G\) and a capacity function \(c\), and the goal is to find a carpool matching \(M\) that maximizes \(|P_M|\).

Given an undirected graph \(G\), let \(D^*\) be a minimum cardinality dominating set, and let \(M^*\) be an optimal carpool matching with respect to \(G\) and the capacity function: \(c(v) = \deg(v)\), for every \(v \in V\).

**Observation 4.** \(|P_{M^*}| + |D^*| = |V|\)

**Proof.** Given a carpool matching \(M\), observe that \(D_M \cup Z_M\) is a dominating set. In the other direction, a dominating set \(D\) induces a carpool matching of size \(|V \setminus D|\). \(\square\)

We use this duality to obtain a hardness result for Maximum Carpool Matching.

**Theorem 22.** The Maximum Carpool Matching problem is APX-hard, even for undirected and unweighted instances with maximum degree bounded by \(b\), for \(b \geq 3\).

**Proof.** We prove the theorem by presenting an \(L\)-reduction from Minimum Dominating Set-b. (For details on \(L\)-reductions the reader is referred to [97].) We define a function \(f\) from Minimum Dominating Set-b instances to Maximum Carpool Matching instances.
as follows: \( f(G) = (G, c) \), where \( c(v) = \deg(v) \), for every \( v \in V \). Next, we define a function \( g \) that given a carpool matching computes a dominating set as follows: \( g(M) = V \setminus P_M \). Both \( f \) and \( g \) can be computed in polynomial time.

Let \( D^* \) be an optimal dominating set with respect to \( G \), and let \( M^* \) be an optimal carpool matching with respect to \( G \) and \( c \). Since \( |D^*| \geq \frac{|V|}{b+1} \), it follows that \( |P_{M^*}| \leq b|D^*| \). In addition, if \( M \) is a carpool matching, we have that

\[
|D_M \cup Z_M| - |P_M| = (|V| - |P_M|) - |P_M| = |P_{M^*}| - |P_M|.
\]

Hence, there is an L-reduction from \textsc{Minimum Dominating Set}-b to unweighted and undirected \textsc{Maximum Carpool Matching} with bounded capacity \( b \).

### 5.3.2 Local Search

In this section we present a local search \((\frac{1}{2} + \frac{1}{2c_{\text{max}}} - \varepsilon)\)-approximation algorithm for unweighted \textsc{Maximum Carpool Matching} whose running time is polynomial if \( c_{\text{max}} = O(1) \).

Let \( k \) be a constant integer to be determined later. Algorithm \textbf{EdgeSwap} (Algorithm 7) maintains a feasible matching \( M \) throughout its execution and operates in iterative manner where in each iteration it tries to find a better solution by replacing a subset of at most \( k \) edges in the current solution with another (larger) subset of edges not in the solution. The algorithm halts when no improvement can be done.

**Algorithm 7: EdgeSwap\((G, c, k)\)**

```plaintext
1  M ← ∅
2  repeat
3    done ← true
4    forall M' ⊆ M : |M'| ≤ k do
5      forall A' ⊆ A \ M : |A'| = |M'| + 1 do
6        if M \ M' ∪ A' is feasible then
7          M ← M \ M' ∪ A'
8          done ← false
9      end
10    end
11  until done;
12  return M
```

Algorithm \textbf{EdgeSwap} terminates in polynomial time, since in every non-final iteration it improves the value of the solution by one. Thus, after at most \( n \) iterations the algorithm terminates. In every iteration the algorithm examines all subsets of edges of a fixed size and tests for feasibility, both these operations can be done in polynomial time.

Observe that a vertex \( v \in D_M \cup Z_M \) is the center of a directed star whose leaves are the passengers in the set \( P_M(v) = \{ u : (u, v) \in M \} \) \((P_M(v) = \emptyset \), for \( v \in Z_M \)). Given a carpool matching \( M \), we define \( S(M) \) to be the set of stars that are induced by \( M \). Denote by \( V(S) \) the set of vertices of a star, i.e., if \( v \) is the center of \( S \), then \( V(S) = \{ v \} \cup P_M(v) \). Also, let \( A(S) \) be the arcs of \( S \). For \( T \subseteq S(M) \), define \( V(T) = \bigcup_{S \in T} V(S) \) and \( A(T) = \bigcup_{S \in T} A(S) \).
It remains to analyze the approximation ratio of **EdgeSwap**. Consider an optimal matching $M^*$, and $M$ be the matching computed by **EdgeSwap**. Given both matchings we build the star graph of the two solutions in which each vertex represents a star from the optimal solution, namely from $S(M^*)$, and an edge exists between two vertices if there is a star in $S(M)$ that intersects the two corresponding stars of the optimal solution. Formally

$$ H = (S(M^*), E) $$

where

$$ E = \{ (S_i^*, S_j^*) : \exists S \in S(M), V(S) \cap V(S_i^*) \neq \emptyset \land V(S) \cap V(S_j^*) \neq \emptyset \} . $$

Figure 5.6 depicts a star graph.

**Lemma 21.** The maximum degree of $H$ is $c_{\text{max}} + 1$.

**Proof.** Each star in $S(M^*)$ contains at most $c_{\text{max}} + 1$ vertices and each such vertex can belong to a star in $S(M)$ containing additional $c_{\text{max}}$ vertices, each of which is located in a different star in $S(M^*)$.

In what follows we compare $|M|$ and $|M^*|$ in maximal connected components of the star graph $H$. Intuitively, we show that $M$ is optimal on small maximal components, and that the approximation ratio on medium (non-necessarily) components can be bounded due to the termination condition of **EdgeSwap**. Large maximal components will be partitioned into medium components.

We first show that large connected graphs (or maximal connected components) can be partitioned into medium size components.

**Lemma 22.** An undirected connected graph $G = (V, E)$ with maximum degree $\Delta$, can be decomposed into connected components of size at least $\ell$ and at most $\Delta \ell$, if $\ell \leq |V|$.

**Proof.** We give a constructive proof. First, if $\ell \leq |V| \leq \Delta \ell$, then we are done. Otherwise, we partition the graph recursively as follows. Let $U = \emptyset$. As long as $|U| < \ell$, choose an arbitrary vertex $u$ such that $U \cup \{u\}$ is connected and add $u$ to $U$. If the graph $G[V \setminus U]$, which is induced by $V \setminus U$, is connected, then choose the next vertex. However, if $G[V \setminus U]$ becomes disconnected, then consider the maximal connected components of $G[V \setminus U]$. Add
the vertices of any maximal component that contains less than ℓ vertices to \( U \). Observe that afterwards \( |U| < \ell + (\Delta - 1)\ell = \Delta \ell \). If \( |U| \geq \ell \), then recursively partition any component of \( G[V \setminus U] \) that contains more than \( \ell \) vertices. If \( |U| < \ell \), then there must exist at least one maximal component \( C \) of \( G[V \setminus U] \) that contains more than \( \ell \) vertices. In this case recursively partition any maximal component of \( G[V \setminus (U \cup C)] \) that contains more than \( \ell \) vertices. Also, continue to augment \( U \) in the graph \( G[U \cup C] \).

Define \( \deg_M(v) \triangleq \deg_{\text{in}}^M(v) + \deg_{\text{out}}^M(v) \). For a subset \( U \subseteq V \) of vertices define \( \deg_M(U) \triangleq \sum_{v \in U} \deg_M(v) \). Observe that \( |M| = \frac{1}{2} \deg_M(V) \).

In the next lemma we bound the degree ratio in a component that contains stars with at most \( k \) arcs.

**Lemma 23.** Let \( \mathcal{T} \subseteq S(M^*) \) that induces a connected subgraph of \( H \). If \( |A(\mathcal{T})| \leq k \), then

\[
\frac{\deg_M(V(\mathcal{T}))}{\deg_{M^*}(V(\mathcal{T}))} \geq \frac{1}{2} + \frac{1}{2c_{\max}} - \frac{1}{2c_{\max}|\mathcal{T}|}.
\]

**Proof.** Consider the solution \( M' \) obtained from \( M \) by removing all the edges from \( M \) that intersect \( V(\mathcal{T}) \) and adding all the edges from \( M^* \) that intersect \( V(\mathcal{T}) \). Observe that if an edge \((u, v)\) in \( M^* \) intersects \( V(\mathcal{T}) \), then \( \{u, v\} \in V(\mathcal{T}) \) by the definition of the graph \( H \). Hence, \( M' \) is feasible carpool matching.

Since \( \mathcal{T} \) induces a connected subgraph of \( H \), the removal of edges in \( M \) that intersect \( V(\mathcal{T}) \) decreased \( |M| \) by at most \( \deg_M(V(\mathcal{T})) - |\mathcal{T}| + 1 \). On the other hand, the increase in size is exactly \( \frac{k}{2} \deg_{M^*}(V(\mathcal{T})) \leq c_{\max} |\mathcal{T}| \). Since \( |A(\mathcal{T})| \leq k \), we know that this difference can not be positive, or else, EdgeSwap would not have terminated. Thus

\[
\frac{k}{2} \deg_{M^*}(V(\mathcal{T})) \leq \deg_M(V(\mathcal{T})) - |\mathcal{T}| + 1,
\]

and so

\[
\frac{\deg_M(V(\mathcal{T}))}{\deg_{M^*}(V(\mathcal{T}))} \geq \frac{1}{2} + \frac{|\mathcal{T}| - 1}{2c_{\max} |\mathcal{T}|} = \frac{1}{2} + \frac{1}{2c_{\max}} - \frac{1}{2c_{\max}|\mathcal{T}|},
\]

as required. \( \square \)

It remains to bound the approximation ratio of EdgeSwap.

**Lemma 24.** If \( k \geq c_{\max} \), then \( |M| \geq (\frac{1}{2} + \frac{1}{2c_{\max}} - \frac{c_{\max}c_{\max}+1}{2k}) \cdot |M^*| \).

**Proof.** Consider a maximal (with respect to set inclusion) connected component of \( H \) induced by the vertices in \( \mathcal{T} \subseteq S(M^*) \). If \( |M \cap A(\mathcal{T})| \leq k \), then it must be that \( |M \cap A(\mathcal{T})| = |M^* \cap A(\mathcal{T})| \), since otherwise \( M \cap A(\mathcal{T}) \) could be improved.

It remains to consider a maximal component \( \mathcal{T} \) such that \( |M \cap A(\mathcal{T})| > k \). Since the number of edges in \( S(M^*) \) is at most \( c_{\max} \), it must be that \( |V(\mathcal{T})| > \frac{k}{c_{\max}} \). Due to Lemma 23 (with \( \ell = \frac{k}{c_{\max}(c_{\max}+1)} \)) we can partition \( \mathcal{T} \) into connected components each of which contains between \( \frac{k}{c_{\max}(c_{\max}+1)} \) and \( \frac{k}{c_{\max}} \) vertices. Since each such vertex set \( \mathcal{X} \) is connected and contains at most \( \frac{k}{c_{\max}} \) stars, it follows that \( |A(\mathcal{X})| \leq k \). Due to Lemma 23 we have that

\[
\frac{\deg_M(V(\mathcal{X}))}{\deg_{M^*}(V(\mathcal{X}))} \geq \frac{1}{2} + \frac{1}{2c_{\max}} - \frac{c_{\max}(c_{\max}+1)}{2k}.
\]
Since $\deg_{M^*}(V(T)) = \sum_{\mathcal{X}} \deg_{M^*}(V(\mathcal{X}))$ and $\deg_M(V(T)) = \sum_{\mathcal{X}} \deg_M(V(\mathcal{X}))$, it follows that
\[
\frac{\deg_M(V(T))}{\deg_{M^*}(V(T))} \geq \frac{1}{2} + \frac{1}{2c_{\text{max}}} - \frac{c_{\text{max}}(c_{\text{max}} + 1)}{2k},
\]
and thus
\[
|M \cap A(T)| \geq \left( \frac{1}{2} + \frac{1}{2c_{\text{max}}} - \frac{c_{\text{max}}(c_{\text{max}} + 1)}{2k} \right) |M^* \cap A(T)|,
\]
and the lemma follows. \hfill \Box

By setting $k = \lceil c_{\text{max}}(c_{\text{max}} + 1)/2\varepsilon \rceil$, we get the following result.

**Corollary 4.** There exists a $(\frac{1}{3} + \frac{1}{2c_{\text{max}} - \varepsilon})$-approximation algorithm for unweighted Maximum Carpool Matching, for every $\varepsilon > 0$.

We now show that Corollary 4 is tight. Consider the example in Figure 5.7; the example is for the special case when $k = 11$ and $c_{\text{max}} = 2$ but this example can be generalized in a straightforward manner. One can verify that as the graph in the example grows, the ratio between the optimal solution and the local search solution approaches $\frac{3}{4}$.

### 5.4 Group Carpool

We now consider Maximum Group Carpool Matching which a variant of Maximum Carpool Matching in which we are given a size function $s : V \rightarrow \mathbb{N}$, and the constraint $\deg_{M^*}^{\text{in}}(v) \leq c(v)$ is replaced with the constraint $\sum_{u:(u,v) \in M} s(u) \leq c(v)$. 

Figure 5.7: A tight example for Corollary 4. The optimal solution is given by the red solid arcs while the local search solution is given by the green, dashed arcs. The solution can be improved by the local search algorithm only if it removes all the edges.
Figure 5.8: An unweighted Maximum Carpool Matching instance. Capacities are written inside vertices, and arcs are labeled with their size. We have that $f(A) + f(B) = 2 < 3 = f(A \cup B) + f(A \cap B)$.

We start by showing that this variant of the problem does not fit the submodular maximization formulation as defined for Maximum Carpool Matching. Recall the submodular maximization formulation given in Section 5.2.1 namely \( \bar{w} : 2^V \to \mathbb{R} \), where \( \bar{w}(S) \triangleq w(M(S)) \) and \( M(S) \) is the maximum weight carpool matching that satisfies \( D_M(S) \subseteq S \subseteq V \setminus P_M(S) \). Figure 5.8 contains an instance that shows that the function \( \bar{w} \) is not submodular anymore.

We show, however, that Maximum Group Carpool Matching has a \((\frac{1}{2} - \varepsilon)\)-approximation algorithm by extending the local improvement algorithm from Section 5.2.2. The main concern when trying to adopt the algorithm to Maximum Group Carpool Matching is how to determine if a vertex can be improved. With Maximum Carpool Matching, if weights are polynomially-bounded, it was enough to consider the incoming arcs to a vertex \( v \) in a non-increasing order of \( \delta_M \) (See proof of Theorem 20). This does not work anymore, since in the Maximum Group Carpool Matching we have sizes. In fact, given \( v \), finding the best star with respect to \( \delta_M \) is a Knapsack instance where the size of the knapsack is \( c(v) \). If weights are polynomially-bounded, then \( \delta_M(e) \) is bounded for every arc \( e \in A \), and therefore this instance of Knapsack can be solved in polynomial time using dynamic programming.

**Theorem 23.** Algorithm **StarImprove** is a \( \frac{1}{2} \)-approximation algorithm for Maximum Group Carpool Matching, if edge weights are integral and polynomially bounded.

Using standard scaling and rounding we obtain the following result.

**Theorem 24.** There exists a \((\frac{1}{2} - \varepsilon)\)-approximation algorithm for Maximum Group Carpool Matching, for every \( \varepsilon \in (0, \frac{1}{2}) \).

Finally, we show that a variant of Algorithm **EdgeSwap** from Section 5.3.2 can be used to solve Maximum Group Carpool Matching while keeping the same approximation guarantees. The only difference is that when checking feasibility of a set of arcs we do not compare the number of passengers to the capacity of a driver, but rather compare the total size of the passengers to the capacity.

**Theorem 25.** There exists a \((\frac{1}{2} + \frac{1}{2 \cdot \text{max}} - \varepsilon)\)-approximation algorithm for unweighted Maximum Group Carpool Matching, for every \( \varepsilon > 0 \).
Chapter 6

Conclusion

In this thesis we developed approximation algorithms for four NP-hard combinatorial optimization problems. Let us now summarize our results.

For the 2-Convex Recoloring problem we developed a greedy $3/2$-approximation algorithm. We show that if the input graph is a path then this is in fact a $5/4$-approximation algorithm. This is the first time a constant-ratio approximation algorithm is given for a variant of CR in general graphs.

Next, we defined a model for service chain placement in SDNs. We show that the placement problem is computationally hard even in very simple cases. Given these hardness results, we propose an FPTAS for the placement problem in acyclic physical networks. For general networks we propose a randomized algorithm and an fixed parameter algorithm whose running time depends on the topology of the network, more specifically on the number of nodes in the network with more than two neighbors. Finally, we extend our results to a fault tolerant setting.

For Submodular Knapsack we developed a framework that for any $\alpha < \beta < \frac{1}{2}$ and an $\alpha$-approximation algorithm $A$ with a running time $t(n)$ produces a $\beta$-approximation algorithm $B$. The running time of $B$ is $O(n^2) + O(t(n))$ where the constants hidden by the big $O$ depend on $\alpha$ and $\beta$. This framework can be used, for example, to develop a $(1 - e^{-2/3})$-approximation algorithm that runs in time $O(n^2)$ with very small constant.

Finally, we show that the Maximum Carpool Matching can be formulated as a submodular optimization problem, thus, admitting a $1/2$-approximation algorithm.
Bibliography


[93] Joseph (Seffi) Naor. personal communication.


71


 FHA. To achieve this, we first enumerate all possible assignments to a set of binary variables and then select the assignment that minimizes the objective function.

In addition, we adopt a heuristic for the general case. The heuristic is proposed as a solution to the problem of minimizing the objective function.

The heuristic is a linear relaxation of the problem, as it simplifies the objective function while still preserving its critical properties.

The problem is NP-hard, and we adopt an efficient algorithm for solving it. The algorithm is a dynamic programming approach that iteratively selects the best assignment.

The algorithm runs in time $O(n^3)$, where $n$ is the number of nodes in the graph. However, in practice, the algorithm is faster due to the linear relaxation.

The algorithm is based on the concept of a minimum cut in a directed graph. The minimum cut is the smallest cut in the graph that separates the source from the sink.

The cut is defined as the sum of the weights of the edges that cross the cut. The algorithm iteratively selects the edge with the smallest weight that crossing the cut and then updates the cut accordingly.

The algorithm is a heuristic, and it is not guaranteed to find the optimal solution. However, it provides a good approximation of the optimal solution.

The algorithm is implemented as a C++ program and tested on a set of benchmark instances. The results show that the algorithm is efficient and provides good solutions.

The algorithm is suitable for solving large-scale instances and can be applied to other problems with similar structures.
The thesis presents a collection of combinatorial optimization problems. The first problem is the graph coloring problem, which is a fundamental problem in graph theory. The coloring of a graph is a function $C$, and a set of colors $G = (V; E)$ is given. The goal is to find a coloring of the graph such that no two adjacent vertices have the same color. The second problem is the graph matching problem, which is a classic problem in computer science. The goal is to find a maximum matching in a graph, which is a set of edges that do not share any vertices.

From the graph coloring problem, we can define the graph matching problem. The coloring of a graph is a function $C$ that assigns a color to each vertex. The goal is to find a coloring of the graph such that the number of vertices with the same color is minimized. The third problem is the graph matching problem, which is a classic problem in computer science. The goal is to find a maximum matching in a graph, which is a set of edges that do not share any vertices.

The fourth problem is the graph matching problem, which is a classic problem in computer science. The goal is to find a maximum matching in a graph, which is a set of edges that do not share any vertices.

The fifth problem is the graph matching problem, which is a classic problem in computer science. The goal is to find a maximum matching in a graph, which is a set of edges that do not share any vertices.

To solve these problems, the thesis proposes an algorithm based on the graph coloring problem. The algorithm works as follows: first, we choose a color for each vertex such that no two adjacent vertices have the same color. Then, we choose a set of colors for the graph such that the number of vertices with the same color is minimized. Finally, we find a maximum matching in the graph, which is a set of edges that do not share any vertices.

These problems have many applications, including in computer science, biology, and other fields. They are also important for understanding the structure of graphs and networks.

The thesis concludes with a discussion of the future work, including the development of new algorithms and the application of these algorithms to real-world problems.
אלגוריתמי קירוב لمแถס של פונקציות
תח-מודלוריות וביעיון תכונת רשתות

הינמר על מחקר

לשם مليי חלקי של הדרישות לрабלה התואר דוקטור לפילוסופיה

גלעד קותיאל

הנה לסטנט הטכנולוגיה – מכון טכנולוגי לישראל
אדר ב' תשע"ט חיפה מפרץ 2019
אלגוריתמי קירוב למקסימום של פונקציה תת-מודולרית ובאיות חכמוני רשתות

גדלד קוטיאל