Optimal Connected Vertex Clustering

Master Thesis in Computer Science

Sebastian Krott

sebastian.krott@rwth-aachen.de
RWTH Aachen University
student ID: 314056

August 20, 2018

1st examiner	2nd examiner
Prof. Dr. Gerhard Woeginger	Prof. Dr. Marco Lübbecke
Chair of Computer Science 1	Chair of Operations Research
Algorithms and Complexity	

Abstract

We introduce an optimization problem on graphs called *Connected Vertex Clustering Problem (CVCP)*. The input consists of a finite graph, arbitrary linear constraints to restrict the set of feasible clusters and a linear objective function. The expected solution is a vertex clustering that optimizes the objective under the condition that each cluster induces a connected subgraph and satisfies the custom constraints. Besides partitional clustering, i.e., node partitioning, the solution may also be restricted to packings or coverings of nodes. We show that this highly configurable problem is NP-hard and propose a *branch-and-price method* as a solution approach. The suggested method is implemented as a *framework* which is capable of solving arbitrary CVCP instances. The framework can easily be extended with new features due to its plug-in architecture. This allows to exploit the characteristics of specific variants of the CVCP in order to enhance the efficiency of the solution process. We evaluate the developed framework on a districting problem for the German federal elections and on the Odd Cycle Packing Problem.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Definition
BWG	Bundeswahlgesetz (German Federal Elections Act)
CVC	Connected Vertex Clustering
CVCP	Connected Vertex Clustering Problem
GI	Greedy Initializer
GP	Greedy Pricer
GPDP	German Political Districting Problem
LP	Linear Program
LPR	LP Relaxation
MIP	Mixed Integer Program
MP	Master Problem
OCPN	Odd Cycle Packing Number
OCPP	Odd Cycle Packing Problem
PR	Pricing Round
RMP	Restricted Master Problem
SIP	Separator Inequalities Pricer
SPSP	Shortest Path Subtrees Pricer
SSP	Stable Set Problem
TCI	Three-Cycle Initializer
TCP	Three-Cycle Pricer

1. Introduction

The digitalization of industries and everyday life, the expansion of the internet and other technological advances like the smart phone have led us to the age of big data. An enormous volume of data is being generated, transmitted, stored and processed on a per-second basis. In this environment, cluster analysis as a tool for data structuring, exploration and processing has become more relevant than ever. A clustering algorithm groups a set of data objects into different clusters based on some notion of similarity. Today, such clustering techniques are used for a plethora of computer science applications including information retrieval, data reduction and pattern recognition.

Part of the ever-growing mass of data like the structure of the web or contacts in online social networks can be represented by graphs. A graph consists of nodes which embody objects and of edges that describe relationships or links between these objects. Considering, e.g., a social network, we may represent users in terms of nodes and let two users be connected by an edge if they are a contact of each other. Since graphs are commonly used as an abstract model, some algorithms have been specifically tailored to solve clustering problems on graphs. Regarding social networks, e.g., clustering algorithms are used for community detection and recommender systems.

Due to the described developments in information technology, recent research in cluster analysis has put a strong emphasis on algorithmic efficiency. Many common large-scale applications have numerous highly efficient algorithms designed specifically for them. However, in many smaller problem scenarios, efficiency is, albeit still desirable, a far less critical factor. Instead, it may be preferable to employ a method that is capable of handling additional cluster requirements that are unique to the problem at hand; or to optimize the clustering based not only on similarity, but also other measures.

This thesis proposes a highly adaptable graph clustering approach that is based on mixed integer programming. Typically, graph clustering algorithms derive the similarity between nodes from the graph's edges or corresponding edge weights. The goal is to obtain dense clusters where all nodes are closely connected to one another. Instead, we consider connectivity as a necessary condition for any cluster and compute a solution that optimizes a completely custom-definable linear objective. The objective may take cluster density into account, but can also consider other factors or even disregard the structure of the graph completely. Moreover, our method allows to impose individual constraints on the clusters via linear inequalities. Thus, certain clusters can be declared infeasible to meet application requirements.

1.1. Motivation

The development of the new clustering approach was inspired by a practical application. Every four years, the German citizens elect the Bundestag, their federal parliament.

Prior to each election, the legislator and the Federal Election Commissioner (German: Bundeswahlleiter) are responsible for redefining the constituencies. Therefore, various aspects regarding administrative boundaries, population deviation, contiguity and the previous constituency allocations are taken into account. To date the constituencies are still defined manually in a nontransparent process. Determining the constituencies in terms of a mathematically well-defined allocation problem would in contrast allow for an unbiased and transparent solution adhering to highest democratic standards. This applies even more to countries without a voting system of proportional representation where *the winner takes it all*. Due to this concept, the outcome of an election may be strongly affected by the definition of constituencies. In the United States of America, the deliberate manipulation of voting districts even reached such extremes that it coined the term "Gerrymandering" which is a blend of the name *Gerry* and the word *salamander* [8, 13, 51]. The administration of Elbridge Gerry, a former Governor of Massachusetts, formed such a bizarre constituency that caricatures compared its shape to a salamander. In addition to preventing purposeful manipulation or unintended minor inequalities, a formal problem definition also creates the potential of saving resources via automatic computation of the constituencies.

Besides, due to changes in the landscape of political parties and peculiarities of the German voting system, the number of members of the German Bundestag has steadily grown over the past legislative periods. In order to cope with this trend, a reduction of the number of constituencies is being discussed [7, 46]. However, in practice it is typically attempted to adjust constituencies as little as possible with respect to the previous election. Consequently, a complete redefinition that optimizes different quality metrics is much more complex than the usually applied "minimum change" allocation strategy. Considering the large amount of potential allocations, possibly even for multiple scenarios with differing numbers of constituencies, an automated approach would once again be not only more efficient but also more traceable.

1.2. Contribution

Originating from the *German Political Districting Problem (GPDP)*, a formal graph-based definition of the constituency allocation problem, we derive a generalized problem that can be applied to many application scenarios. This *Connected Vertex Clustering Problem (CVCP)* is defined as a mixed integer program which allows the introduction of custom variables, custom linear constraints and a custom linear objective function. Additionally, the CVCP allows to define which type of clustering should be computed, i.e., whether nodes shall occur in exactly, at least or at most one cluster.

We show that the CVCP is NP-hard and develop a branch-and-price method that solves arbitrary problem instances to optimality. For this purpose, we extend the so-called Ryan-Foster branching for partitioning problems to packing and covering scenarios.

Our method is implemented as a framework that serves as a general solver and is designed for extendability. Thus, it is possible to increase efficiency by exploiting special characteristics of different CVCP variants. The developed framework is evaluated on the GPDP and the *Odd Cycle Packing Problem (OCPP)*, another optimization problem on graphs. We compare the computational results for different general purpose and problem-specific components.

1.3. Outline

After the introduction, Chapter 2 defines certain terminology and notations to provide a basic understanding for the subsequent content. It touches on different areas like graph theory, cluster analysis and linear and mixed integer programming. In order to go more into depth on topics closely related to this thesis, Chapter 3 discusses some of the related work. Chapter 4 formally introduces the original formulation of the CVCP as a mixed integer program. Using Dantzig-Wolfe decomposition, the CVCP's original formulation is then transformed into an aggregated extended formulation. In Chapter 5, we propose a branch-and-price approach for solving CVCP instances. For this purpose, we generalize Ryan-Foster branching to the packing and covering case. The master problem, the pricing problem, the branching and some additional aspects are described in detail. Chapter 6 presents the *Connected Vertex Clustering (CVC)* Framework that we developed for evaluating our approach. After discussing the framework architecture and selected features, we examine different implementations for modeling connectivity through linear constraints. Chapters 7 and 8 introduce the OCPP and the GPDP as specializations of the CVCP. Each problem is formally defined and problem-specific plug-ins for the CVC Framework are presented. To evaluate our method, Chapter 9 discusses computational results that are obtained from applying different variants of our solving approach to OCPP and GPDP instances. Finally, Chapter 10 summarizes all findings in a conclusion and provides an outlook on potential future research.

2. Preliminaries

For more clarity and a better understanding, we introduce basic terms and notations.

2.1. Common Math

For $n \in \mathbb{N}$, we denote the set $\{1, \ldots, n\}$ of the first n natural numbers as $[n]$. The *power set* $P(S)$ of a set S is the set of all subsets of S, i.e., $P(S) = \{S' \mid S' \subseteq S\}$. It holds $|P(S)| = 2^{|S|}$. Assume that the set S is numbered, i.e., $S = \{s_1, ..., s_n\}$. The *characteristic vector* $\boldsymbol{x}_{S'} \in \{0,1\}^n$ of subset $S' \subseteq S$ is then given by $(x_{S'})_i = 1$ iff $s_i \in S'$. Two arbitrary sets S_1 and S_2 are called *disjoint*, if they do not share any elements, i.e., if $S_1 \cap S_2 = \emptyset$. For some vector $\boldsymbol{x} \in \mathbb{R}^n$, we define its *support* as the set of all indices whose components differ from zero, i.e., $supp(\boldsymbol{x}) = \{i \mid x_i \neq 0\}$. For $S' \subseteq S$, it follows $S' = \{s_i \in S \mid i \in supp(\boldsymbol{x}_{S'})\}$.

2.2. Graph Theory

We will continue with a brief introduction into graph theory. A *(finite) graph* $G = (V, E)$ is defined as a pair consisting of a set of *nodes* or *vertices* $V = \{v_1, \ldots, v_n\}$ and a set of *edges* $E = \{e_1, \ldots, e_m\}$. When dealing with multiple graphs, the node set and edge set of a graph G are also denoted as $V(G)$ and $E(G)$, respectively, to avoid confusion. Evidently, the number of nodes is given by $|V| = n \in \mathbb{N}$ and the number of edges by $|E| = m \in \mathbb{N}$.

One distinguishes between *directed* and *undirected* graphs. If a graph G is directed, then each edge $e \in E$ is a pair of two nodes, i.e., $e = (u, v)$ with $u, v \in V$. Through the pair's order the edge is given a specific orientation. We can say that e is directed from the source (node) u to the target (node) v. On the other hand, if G is undirected, then an edge $e = \{u, v\}$ is simply a set of two nodes and the edge does not possess any orientation.

If a node u belongs to an edge e, it is also referred to as an *end node* of that edge. Two nodes $u, v \in V$ which are connected by an edge $e \in E$ are said to be *neighbors* or *adjacent* to one another. Additionally, we say that u and v are *incident* to e and vice versa. Two edges sharing a common end node are *incident*, too.

An edge with two identical end nodes is called a loop. A simple graph is an undirected graph without loops. Unless stated otherwise, we will assume graphs to be simple and finite.

Definition 2.1 (Subgraph, Node-Induced Subgraph)**.** A graph G′=(V′,E′) is a *subgraph* of G=(V,E), iff $V' \subseteq V$ and $E' \subseteq E$. Since G′ is a graph, V' must contain all end nodes of the edges E'. If E' contains all the edges from E whose end nodes are both in V', i.e., $E' = \{e \in E \mid e \subseteq V'\}$, then we refer to G' as a *(node-)induced subgraph*. Given a graph G, any node-induced subgraph is defined by the node set V' alone and therefore denoted $G[V']$.

Definition 2.2 (Walk, Finite Path, Connectivity)**.** A *(finite) walk* p of length ℓ in an undirected graph G is a sequence of ℓ of its edges $(e_{j_1}, \dots, e_{j_\ell})$ connecting a sequence of $\ell + 1$ vertices $(v_{i_1}, \dots, v_{i_{\ell+1}})$. This means that $e_{j_k} \in E$ is incident to $v_{i_k} \in V$ and $v_{i_{k+1}} \in V$ for all $k \in [\ell]$. A *path* is a walk p without repeating nodes. An undirected graph G is *connected* iff each pair of nodes $u, v \in V$ is connected through a path.

In order to express, e.g., distances between nodes, the edges of a graph can be assigned *edge weights* $w(e) \in \mathbb{R}$. For such a *weighted graph*, we define a path's length as the sum of the weights of the corresponding edges, i.e., $\ell(p) = \sum_{e \in p} w(e)$. A path from a node u to a node v is a *shortest path*, if there exists no path of smaller length connecting these nodes. The *distance* $d(u, v)$ between two nodes is defined as the length of a shortest path from u to v. If the two nodes are not connected and no such path exists, it holds $d(u, v) = \infty$. An unweighted graph is transformed into a weighted graph by defining $w(e) = 1$ for all edges $e \in E$.

Definition 2.3 (Cycle, Simple Cycle, Tree)**.** A *closed walk* or *cycle* is a walk $c = (v, \dots, v)$ where the first and the last node are the same. If a cycle has no further repetition of nodes other than at the last node, it is called a simple cycle. A tree is a connected graph without cycles.

A *subtree* is a tree subgraph of a graph G. Note that each pair of nodes in a tree is connected by exactly one path. A *shortest path subtree* of G with root $r \in V'$ is a subtree where each path $(r, \dots, v)$ from the root to a node $v \in V'$ is a shortest path in G.

2.3. Cluster Analysis

As already mentioned in Chapter 1, clustering is the task of dividing a set $S = \{s_1, \dots, s_n\}$ of $n \in \mathbb{N}$ objects into different groups or clusters [22, 61]. These clusters should be homogeneous, i.e., contain objects that share common characteristics or are otherwise closely related to one another. The resulting set of clusters $\mathcal{C} = \{C_1, C_2, \dots\}$ with $C_k \subseteq S$ is also called clustering. Each object must occur in at least one subset, i.e., it must hold $\bigcup_k C_k = S$. If all clusters are pairwise disjoint, then $\mathcal{C}$ is called a partitional clustering. If objects may occur in multiple clusters, we refer to $\mathcal{C}$ as an overlapping clustering. In this thesis, we examine the clustering of graph nodes, where the objects S correspond to the nodes V. In this context we will use the term cluster not only to refer to subsets $C_k \subseteq V$ of some clustering $\mathcal{C}$, but also to the corresponding subgraphs $G[C_k]$.

In order to group the objects, a clustering algorithm typically makes use of a similarity function $s(s_i, s_j)$ which assigns a similarity value to an arbitrary pair of objects $s_i, s_j \in S$. For vertex clustering, the similarity of two nodes $u, v \in V$ can be derived from their distance, i.e., $s(u, v) = \frac{1}{d(u,v)}$. However, in contrast to many other graph clustering approaches, the aim of this thesis is not to determine clusters of minimal node distances. Instead, we regard connectivity as a necessary condition for each cluster, but do not prioritize density, i.e., the amount of edges within the clusters.

2.4. Linear Programming

A *linear program (LP)* is a formal representation of an optimization problem in terms of linear constraints and a linear objective function [50]. An LP in *canonical form* is defined as follows:

$$\begin{aligned} \max\quad & \boldsymbol{c}^T\boldsymbol{x} && (2.1)\\ \text{s.t.}\quad & A\boldsymbol{x} \leq \boldsymbol{b} && (2.2)\\ & \boldsymbol{x} \geq 0 && (2.3) \end{aligned}$$

Here, the term (2.1) is the *linear objective function* that is to be maximized over the variables $\boldsymbol{x} \in \mathbb{R}^n$ for given coefficients $\boldsymbol{c} \in \mathbb{R}^n$. Inequality (2.2) models the *linear constraints* that are to be satisfied. They are defined by a given coefficient matrix $A \in \mathbb{R}^{m \times n}$ as the left hand side and a right hand side $\boldsymbol{b} \in \mathbb{R}^m$. Moreover, inequality (2.3) adds non-negativity constraints for all variables.

A solution (vector) $\boldsymbol{x}$ is called *feasible*, iff it satisfies both constraint (2.2) and constraint (2.3). Otherwise, $\boldsymbol{x}$ is *infeasible*. A feasible solution $\boldsymbol{x}^*$ is *optimal* iff

$$\boldsymbol{c}^T\boldsymbol{x}^* = \max\,\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \text{ is feasible}\} \quad .$$

An LP is *feasible* if it has a feasible solution and *infeasible* otherwise. A feasible LP without an optimal solution is called *unbounded* since the objective function may then assume arbitrarily large values.

The main solution approach for LPs is the *simplex method* [15]. This method iteratively selects new feasible solutions of monotonously increasing objective value. Given the right configuration, the algorithm is finite and has proven to perform well for many practical applications despite exponential worst case complexity.

For any LP, there exists a corresponding so-called *dual problem*. Given a canonical LP as above, the dual problem is defined as:

$$\begin{aligned} \min\quad & \boldsymbol{b}^T\boldsymbol{y}\\ \text{s.t.}\quad & A^T\boldsymbol{y} \geq \boldsymbol{c}\\ & \boldsymbol{y} \geq 0 \end{aligned}$$

In the context of duality, the original problem is also referred to as the primal problem. Note that the dual problem is an LP, too. Any feasible solution of the dual problem provides a bound for the objective function value of feasible primal solutions. Assume, e.g., that the objective function of the primal problem is to maximized. Let $\boldsymbol{x}$ be a feasible solution of the primal and $\boldsymbol{y}$ a feasible solution of the dual problem. Then it holds

$$\boldsymbol{c}^T\boldsymbol{x} \leq \boldsymbol{b}^T\boldsymbol{y} \quad . \tag{2.4}$$

An objective function bound that is derived from the dual problem according to inequality (2.4) is named *dual bound*.

2.5. Mixed Integer Programming

Mixed integer (linear) programming extends linear programming techniques to a more generalized form of problems [50]. In contrasts to LPs, a *mixed integer program (MIP)* may additionally comprise integer constraints for each variable x_i with $i \in [n]$ such as $x_i \in \{0,1\}$ or $x_i \in \mathbb{N}$. Hence, MIPs extend the range of expressible problems. The *LP relaxation (LPR)* of a MIP is the LP that is obtained by replacing all integer constraints with corresponding lower and upper bounds for the variables. E.g., for a binary variable one would define 0 as a lower and 1 as an upper bound.

In order to solve MIPs, typically the so-called *branch-and-bound* approach is applied which solves multiple subproblems in a tree-like structure [14]. We start with the original problem as the root node of the *branch-and-bound tree*. If the optimal solution of the LPR at the current node is fractional, we split the corresponding problem into multiple subproblems by adding new constraints according to some branching rule. The subproblems are then added to the branch-and-bound tree as child nodes of the current node. If the optimal solution of the LPR at the current node is integral or the LPR is infeasible, then the MIP of the node is solved as well. Hence, the corresponding branch terminates, rendering the node as a leaf of the final branch-and-bound tree. For any non-leaf node in the branch-and-bound tree, an optimal solution of its MIP is given by the best optimal solution of any of its child nodes. If all children are infeasible, then so is the parent node. Consequently, once every branch terminated, the solution to each MIP and in particular the original root node MIP can be computed bottom-up.

The branching rule must meet some criteria to ensure that the branch-and-bound approach terminates and finds an optimal solution. First of all, if the parent node MIP has an optimal solution, then at least one optimal solution must be feasible in one of the child nodes, too. This is guaranteed to be true in particular when each feasible solution of the parent MIP is feasible in at least one of the child nodes. Secondly, all branches must terminate, i.e., reach a leaf node where the LPR either has an integral solution or is infeasible.

One common branching rule is to divide the parent MIP into two subproblems, each with either a new upper or lower bound for some variable with fractional optimal solution value $x_i^* \notin \mathbb{N}$ [1]:

$$\begin{aligned} x_i &\leq \lfloor x_i^* \rfloor \\ x_i &\geq \lceil x_i^* \rceil \end{aligned}$$

This strategy is also referred to as variable branching.

Moreover, it is sometimes possible to terminate branches early. The current node in the branch-and-bound tree is then not further divided into subproblems although its LPR is feasible and the optimal solution still fractional. Assume that we maximize the objective function and that an upper bound $\bar{z}$ is known for the objective of the optimal solution of the node's MIP. This may, e.g., be a dual bound as described in Section 2.4. If a feasible solution $\boldsymbol{x}$ for the original problem with $\boldsymbol{c}^T\boldsymbol{x} \geq \bar{z}$ has already been found at some other node in the branch-and-bound tree, then there is no need to investigate the current branch any further.

2.6. Set Covering, Packing and Partitioning

The CVCP may be regarded as a variant of the problems of set covering, packing and partitioning, which are closely related to one another [3, 38, 57].

2.6.1. Problem Definition

For each of the three problems, the input consists of a finite set $S = \{s_1, ..., s_n\}$ named universe and a set $\mathcal{S} = \{S_1, \dots, S_{n_s}\} \subseteq P(S)$ of $n_s \in \mathbb{N}$ different corresponding candidate subsets. The expected output is a selection $\mathcal{C} \subseteq \mathcal{S}$ of subsets whose cardinality is to be either minimized or maximized under different conditions in each case. For the exact problem definition and later use, we define the following terms:

A set $\mathcal{C} \subseteq P(S)$ is a *covering* of the universe S iff each element of S is contained in at least one subset $S_k \in \mathcal{C}$:

$$\bigcup_{S_k \in \mathcal{C}} S_k = S \tag{2.5}$$

A set $\mathcal{C} \subseteq P(S)$ is a *packing* of the universe S iff no element of S occurs in more than one subset $S_k \in \mathcal{C}$, i.e., all subsets are pairwise disjoint:

$$S_{k_1} \cap S_{k_2} = \emptyset \quad \forall S_{k_1}, S_{k_2} \in \mathcal{C} \tag{2.6}$$

A set $\mathcal{C} \subseteq P(S)$ is a *partitioning* of the universe S iff each element of S occurs in exactly one subset $S_k \in \mathcal{C}$, i.e., if it fulfills both condition (2.5) and condition (2.6).

Note that any partitioning is also both a packing and a covering. Based on the previous definitions we define the three optimization problems as follows:

Definition 2.4 (Set Covering Problem).
Input: A universe S and candidate sets $\mathcal{S} = \{S_1, \dots, S_{n_s}\}$, $S_k \subseteq S$.
Output: A covering $\mathcal{C}^* \subseteq \mathcal{S}$ of minimal cardinality.

Definition 2.5 (Set Packing Problem).
Input: A universe S and candidate sets $\mathcal{S} = \{S_1, \dots, S_{n_s}\}$, $S_k \subseteq S$.
Output: A packing $\mathcal{C}^* \subseteq \mathcal{S}$ of maximal cardinality.

Definition 2.6 (Set Partitioning Problem).
Input: A universe S and candidate sets $\mathcal{S} = \{S_1, \dots, S_{n_s}\}$, $S_k \subseteq S$.
Output: A partitioning $\mathcal{C}^* \subseteq \mathcal{S}$ of minimal cardinality.

For all three optimization problems, the corresponding decision problem is known to be NP-complete.

2.6.2. Mixed Integer Program Formulation

All three problems can be formulated as a MIP. First, we introduce decision variables λ_k:

$$\lambda_k \in \{0,1\} \text{ with } \lambda_k = \begin{cases} 1 & \text{if candidate set } S_k \text{ occurs in the clustering } \mathcal{C} \\ 0 & \text{otherwise} \end{cases} \qquad \forall k \in [n_s]$$

Corresponding coefficients a_{ik} are given by:

$$a_{ik} \in \{0,1\} \text{ with } a_{ik} = \begin{cases} 1 & \text{if object } s_i \text{ is an element of candidate set } S_k \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in [n], \forall k \in [n_s]$$

The MIP is then defined as:

$$\max \quad \sum_{k \in [n_s]} c\lambda_k \tag{2.7}$$

$$\text{s.t.} \quad A\boldsymbol{\lambda} \begin{cases} \leq \mathbb{1} & \text{if set packing} \\ = \mathbb{1} & \text{if set partitioning} \\ \geq \mathbb{1} & \text{if set covering} \end{cases} \tag{2.8}$$

$$\boldsymbol{\lambda} \in \{0,1\}^{n_s} \tag{2.9}$$

Here, $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_{n_s})^T$ is the vector of all decision variables λ_k. In the objective function (2.7), we define $c = 1$ for set packing and $c = -1$ for set covering and partitioning in order to optimize cardinality. Depending on the problem, constraint (2.8) ensures that the derived solution is a packing, partitioning or covering. Constraint (2.9) enforces binary values for the variables.

2.6.3. Related Problems

Besides the standard variants in the Definitions 2.4, 2.5 and 2.6, each of the three problems may be generalized for cost optimization. For this purpose, all subsets $S_k \in \mathcal{S}$ are associated with a cost $c(S_k)$ as an additional input. Instead of maximizing or minimizing the cardinality of $\mathcal{C}$ the goal is then the optimization of the total cost $c(\mathcal{C}) = \sum_{S_k \in \mathcal{C}} c(S_k)$. Typically, it holds $c(S_k) > 0$ for set packing and $c(S_k) < 0$ for set partitioning. The standard variant of each problem is the special case where the cost function is constant so that no set is given any preference.

Furthermore, the problems of set selection are closely related to vertex clustering. Assume that the universe S corresponds to the node set V. Consequently, the candidate subsets $\mathcal{V}$ are node clusters $V_k \subseteq V$. The Set Partitioning Problem is then equivalent to computing a partitional clustering of minimal cost since each node appears in exactly one subset of a feasible solution $\mathcal{C}$. Similarly, the Set Covering Problem corresponds to searching a minimal cost overlapping clustering since each node must be part of at least one subset in the solution. Note at last that according to Definition 2.5, a solution to the Set Packing Problem is not necessarily a clustering because a packing does not always contain all elements of the universe. However, any node packing can easily be extended to a clustering by adding one additional set that comprises exactly the vertices that were left out. Given this interpretation, in the following we apply the term clustering also to node packings.

3. Related Work

3.1. Vertex Clustering

The task of vertex clustering has been studied extensively and numerous general clustering approaches as well as methods designed specifically for clustering vertices are available [2, 36, 49]. Some of these methods are highly efficient and thus applicable to large graphs with hundreds of thousand or even millions of nodes. Certain approaches already support the computation of connected clusters or even "highly connected clusters" already out-of-the-box [37]. Others, like hierarchical clustering, can be configured to also compute connected clusters only.

However, the suggested problem requires additional restrictions to be taken in to account. *Constrained clustering* algorithms allow to include prior knowledge or conditions into cluster computation [6, 20]. Most of the developed approaches consider two different types of constraints. *Must-link* constraints determine that two different objects $s_i, s_j \in S$ must belong to the same cluster. In contrast, *cannot-link* constraints state that the objects s_i and s_j must be assigned to different clusters. Beyond that, some clustering methods were introduced that obey constraints for balancing the size of clusters or binding it from below [4, 6]. Despite providing some control over cluster feasibility, these techniques are still too inflexible for modeling a wider range of custom constraints and additionally do not allow the optimization of user-defined objective functions.

3.2. Ryan-Foster Branching

For enhanced customizability, we opt for a mixed integer programming approach. This enables the definition of an arbitrary linear objective function to be optimized and the introduction of constraints in terms of linear inequalities. As already explained in Section 2.6, the model of the Set Covering, Packing and Partitioning Problem can be applied to compute different types of clusterings. As a branching rule, it is possible to employ standard variable branching. However, this would directly fix a single decision variable with fractional solution value $0 < \lambda_k^* < 1$ to the value 0 in the first branch and to value 1 in the second branch. Setting λ_k to 0 excludes the corresponding cluster S_k from the clustering. Setting λ_k to 1 forces S_k to be part of the clustering. Generally, in the first case remain a lot more potential solutions than in the second one. The resulting branch-and-bound tree is thus heavily unbalanced, wherefore this branching rule is not recommendable for the CVCP.

An alternative strategy is the so-called *Ryan-Foster branching*, which was developed for set partitioning problems [5, 48, 52, 55]. Let all candidate subsets be different and non-empty. Assume then that the given optimal LP solution $\boldsymbol{\lambda}^*$ contains some decision variable λ_a with a fractional solution value λ_a^*. Since no candidate subset is empty, there exists some object $s_i \in S_a$. Due to the partitioning constraint (2.8), there must be a second candidate subset S_b with $s_i \in S_b$ and $0 < \lambda_b^* < 1$. Moreover, all candidate subsets are different, so there is some other object s_j that appears either in the first subset S_a or in the second subset S_b, but not in both. Consequently, it holds:

$$0 < \sum_{S_k \in \mathcal{S} : \{s_i, s_j\} \subseteq S_k} \lambda_k^* < 1 \tag{3.1}$$

To perform Ryan-Foster branching, we determine two such objects s_i and s_j and divide the problem into two branches based on the following constraints:

$$\sum_{S_k \in \mathcal{S} : \{s_i, s_j\} \subseteq S_k} \lambda_k = 1 \tag{3.2}$$

$$\sum_{S_k \in \mathcal{S} : \{s_i, s_j\} \subseteq S_k} \lambda_k = 0 \tag{3.3}$$

According to equation (3.1), the previous fractional solution $\boldsymbol{\lambda}^*$ is infeasible in both branches. We refer to the branch with the first constraint as the *same*-branch. It excludes all candidate subsets that contain only exactly one of the objects s_i and s_j. W.l.o.g., assume $s_j \in S_b$. The same-branch then enforces $\lambda_a = 0$. The branch with the second constraint is called the *differ*-branch. Here, all subsets comprising both objects s_i and s_j are excluded and thus it must hold $\lambda_b = 0$.

As each object must occur in exactly one selected subset, any feasible MIP solution will satisfy either equation (3.2) or equation (3.3). Moreover, the number of objects and therefore the number of pairs to branch on is finite. Each time we branch, the sum in the inequality (3.1) which is determined by the pair $\{s_i, s_j\}$ is restricted to an integral value. Consequently, after a finite number of branchings no such fractional sum can exist. Since we can derive such a sum for any fractional decision variable, an optimal solution must then be integral.

3.3. Connectivity

Another important aspect is the definition of *connectivity constraints* to ensure connectivity of the node clusters. As a common modeling tool, graphs form the foundation of many famous optimization problems in the area of operation research like the Traveling Salesman Problem, network flow problems or assignment problems. Still, we encountered only a few techniques to express subgraph connectivity via linear constraints.

Let us introduce some new variables for presenting these concepts. For some given graph G, a subset of nodes V' shall be determined that induces a connected subgraph $G[V']$. Whether a node $v_i \in V$ belongs to subset V' is modeled by a decision variable x_i:

$$x_i \in \{0,1\} \text{ with } x_i = \begin{cases} 1 & \text{if } v_i \in V' \\ 0 & \text{otherwise} \end{cases} \qquad \forall v_i \in V$$

Besides the node variables, some approaches employ additional edge variables y_{ij}

$$y_{ij} \in \{0,1\} \text{ with } y_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E' \\ 0 & \text{otherwise} \end{cases} \quad \forall \{v_i, v_j\} \in E \tag{3.4}$$

where $E' = E(G[V'])$ is the edge set of the subgraph induced by V'. In order to ensure that each edge variable y_{ij} assumes value 1 iff both corresponding end nodes v_i and v_j are selected, the following constraints are added:

$$x_i \geq y_{ij} \quad \forall \{v_i, v_j\} \in E \tag{3.5}$$

$$x_j \geq y_{ij} \quad \forall \{v_i, v_j\} \in E \tag{3.6}$$

$$x_i + x_j - 1 \leq y_{ij} \quad \forall \{v_i, v_j\} \in E \tag{3.7}$$

Here, the inequalities (3.5) and (3.6) model the implications

$$y_{ij} = 1 \Rightarrow x_i = 1 \wedge x_j = 1 \quad \forall \{v_i, v_j\} \in E$$

and the constraints (3.7) the reverse implications

$$x_i = 1 \wedge x_j = 1 \Rightarrow y_{ij} = 1 \quad \forall \{v_i, v_j\} \in E \quad .$$

All constraints combined result in the equivalence of both statements. Note that the discreteness of the node variables x_i and the previous constraints imply the discreteness of the edge variables, which therefore is not necessary to demand.

3.3.1. Node Cuts

One option is to formulate connectivity constraints based on *(node) cuts* [24, 41]. A node cut $C = (\tilde{V}, V \setminus \tilde{V})$ is a partitioning of the node set V into two disjoint subsets $\tilde{V} \subseteq V$ and $V \setminus \tilde{V}$. Each node cut determines a corresponding cut set $\tilde{E}(C)$ as the set of all edges with exactly one end node in $\tilde{V}$:

$$\tilde{E}(C) = \{\{v_i, v_j\} \in E \mid v_i \in \tilde{V}, v_j \notin \tilde{V}\}$$

A graph is connected iff for each node cut with $\emptyset \subset \tilde{V} \subset V$ the corresponding cut set is non-empty.

In order to define suitable node cuts for the connectivity constraints, the node set is expanded by an artificial source node, i.e., $V^+ = V \cup \{v_s\}$. New edges are added to connect the source node to all other nodes:

$$E^+ = E \cup \{\{v_s, v_i\} \mid v_i \in V\}$$

Then a node induced subgraph of the original graph is determined which additionally is connected to the source node v_s by a single edge.

The source node is automatically considered to be selected, thus a corresponding node variable is not required. Only additional decision variables y_{si} are introduced to facilitate the selection of a source node edge:

$$y_{si} \in \{0,1\} \text{ with } y_{si} = \begin{cases} 1 & \text{if source node edge } \{v_s, v_i\} \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \quad \forall v_i \in V$$

Connectivity is obtained via two different types of constraints:

$$\sum_{\{v_i,v_j\}|v_i\in(V^+\setminus\tilde{V}),v_j\in\tilde{V}} y_{ij} \geq x_t \quad \forall\tilde{V} \subseteq V, \forall v_t \in \tilde{V} \tag{3.8}$$

$$\sum_{\{v_s,v_j\}\in E^+} y_{sj} = 1 \tag{3.9}$$

Here, equation (3.9) ensures that only one source node edge is selected as intended. The cut inequalities (3.8) guarantee that there is no empty cut set for the derived subgraph.

For a more detailed understanding assume that the derived subgraph is unconnected, i.e., there are two different connected components V_a and V_b. The source node v_s can be connected to at most one of these components due to equation (3.9). W.l.o.g. we assume that v_s is not connected to V_b. For $\tilde{V} = V_b$ and any arbitrary $v_t \in V_b$ constraint (3.8) is then not satisfied.

Assume in reverse that the subgraph is connected. If $\emptyset \subset \tilde{V} \cap V' \subset V'$, then the cut set $\tilde{E}(\tilde{V} \cap V', V' \setminus \tilde{V})$ in $G[V']$ is non-empty and the cut inequality holds for all $v_t \in \tilde{V}$. If $\tilde{V} \cap V' = \emptyset$, the constraint holds because $x_t = 0$ for all $v_t \notin V'$. Finally, if $V' \subseteq \tilde{V}$, then some $v_j \in \tilde{V}$ is connected to the source node. Thus, y_{sj} equals 1 and since $v_s \notin \tilde{V}$ the cut inequality is again satisfied.

3.3.2. Node Separators

Another approach is to define connectivity constraints using *node separators* [12, 25, 59]. Given two non-adjacent nodes $a, b \in V$, an (a,b)-separator is a set of nodes $\tilde{V} \subseteq V \setminus \{a,b\}$ so that there is no path connecting a and b in $G[V \setminus \tilde{V}]$. An (a,b)-separator $\tilde{V}$ is *minimal*, iff there is no proper subset $\tilde{V}_2 \subset \tilde{V}$ that also separates a and b. For $V' \subseteq V$, it holds:

$$\exists \text{ path } p = (a,\ldots,b) \text{ in } G[V'] \Leftrightarrow \forall \text{ minimal } (a,b)\text{-separators } \tilde{V} : \tilde{V} \cap V' \neq \emptyset \tag{3.10}$$

Consequently, $G[V']$ is connected iff the right hand side of equivalence (3.10) is satisfied for all node pairs $a, b \in V'$.

Thus, connectivity is achieved by ensuring that for any given pair of nodes in the subgraph all corresponding minimal a,b-separators $\tilde{V}$ contain at least one node $v \in V'$:

$$\sum_{v_i\in\tilde{V}} x_i \geq x_a + x_b - 1 \qquad \forall v_a, v_b \in V : \{v_a, v_b\} \notin E, \forall \text{ minimal } (v_a, v_b)\text{-separator } \tilde{V} \tag{3.11}$$

If both nodes a and b are selected, the sum to the left must be at least 1 so that all minimal (a,b)-separators contain at least one node that also occurs in V'.

3.3.3. Shortest Path Subtrees

Mehrotra et al. propose a strategy for deriving *heuristic* connectivity constraints from *shortest path subgraphs* [45]. Since their work focuses on the definition of constituencies for the federal elections in the United States of America, one major goal is the prevention of Gerrymandering as described in Section 1.1, i.e., the manipulation of constituencies for affecting election outcomes. Consequently, they suggest that constituencies should be compact and penalize non-compactness d_Σ measured by the reciprocal of the closeness centrality of some center node $v_c \in V'$:

$$d_\Sigma(v_c) = \sum_{v \in V'} d(v, v_c)$$

A constituency or cluster is thus considered compact if all of its nodes can be reached from a given center v_c within short distance. With this objective in mind, constraints are defined that do enforce connectivity but exclude some connected subgraphs that are "unlikely to be compact" according to the chosen center v_c and d_Σ.

For a given center v_c, the idea is to compute a subtree of a shortest path tree with root v_c. For $v \in V$, let

$$N^<(v, v_c) = \{u \in V \mid \{u, v\} \in E \land d(u, v_c) = d(v, v_c) - 1\}$$

be defined as the *shortest path predecessors* of v with respect to v_c, i.e., the set of all nodes that are adjacent to v but closer to v_c. The nodes $\{v_c\} \subseteq V' \subseteq V$ form a shortest path subtree with root v_c iff each selected node $v \in V' \setminus \{v_c\}$ has a selected neighbor that is closer to v_c then itself, i.e.,

$$N^<(v, v_c) \cap V' \neq \emptyset \quad \forall v \in V' \setminus \{v_c\} \quad . \tag{3.12}$$

Connecting each node $v \in V' \setminus \{v_c\}$ to one of the neighbors in $N^<(v, v_c)$ yields a corresponding tree. The node-induced subgraph $G[V']$ is derived from the tree by only adding further edges and is therefore also connected.

Condition (3.12) translates to the following linear constraints:

$$\sum_{v_j \in N^<(v_i, v_c)} x_j \geq x_i \quad \forall v_i \in V \setminus \{v_c\} \tag{3.13}$$

The remaining problem is that the choice of a suitable center node v_c is dependent of the subset V' to be determined. To circumvent this issue, the resulting MIP is solved once for each node $v \in V$ as the center.

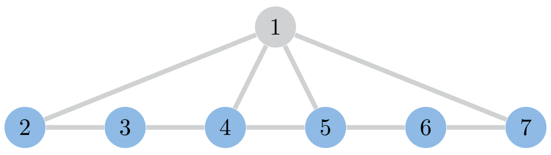


Figure 3.1.: *Connected subgraph (marked blue) without corresponding shortest path subtree.*

Note that condition (3.12) is sufficient but not necessary for connectivity, even if the center v_c can be chosen arbitrarily. Figure 3.1 provides the counter-example of a connected subgraph without a corresponding shortest path subtree. Let V' be the set of all nodes marked blue. Then $G[V']$ is connected, but there is no $v_c \in V'$ that fulfills condition (3.12). For $v_c \in \{2, 3, 4\}$, it holds $N^<(7, v_c) = \{1\}$ and thus $N^<(7, v_c) \cap V' = \emptyset$. In reverse, for $v_c \in \{5, 6, 7\}$, $N^<(2, v_c) = \{1\}$ and again $N^<(2, v_c) \cap V' = \emptyset$.

4. Connected Vertex Clustering Problem

This chapter formally introduces the CVCP as a linear optimization problem. In the first part, we formulate the so-called *original formulation*. This linear program is based on binary assignment variables x_{ik} stating whether a node $v_i \in V$ belongs to cluster $C_k \subseteq V$ of the clustering $\mathcal{C}$. Next, we show that the CVCP is an NP-hard optimization problem. Finally, we apply *Dantzig-Wolfe decomposition* to transform the original formulation into an *aggregated extended formulation*. Here, for each feasible cluster C a decision variable λ_C determines whether $C \in \mathcal{C}$.

4.1. Original Formulation

Section 2.6 already outlined how to model clusterings as MIPs. We follow this approach for the CVCP. The goal is to compute an optimal cost clustering for the nodes of a given graph G. The nodes of each cluster must form a connected subgraph of G. The cost of a clustering is determined by a linear objective function. Arbitrary additional variables and constraints may be introduced for each cluster to adapt the objective function and to define further restrictions besides connectivity.

Formally, we denote the input of the problem as a tuple

$$(o\_type, c\_type, G, Y, \mathcal{Y}, A, \boldsymbol{b}, \boldsymbol{c}_x, \boldsymbol{c}_y, k_{min}, k_{max}) \tag{4.1}$$

of compulsory and optional data. Table 4.1 provides an overview over the inputs.

Table 4.1.: *Input of the CVCP.*

Symbol	Description
o_type	optimization type (*max*/*min*)
c_type	clustering type (*partitioning*/*packing*/*covering*)
G	graph to cluster
Y	(*optional*) matrix of additional custom variables
$\mathcal{Y}$	integer constraints on Y
$A, \boldsymbol{b}$	(*optional*) matrix and vector defining additional custom constraints
$\boldsymbol{c}_x$	assignment costs
$\boldsymbol{c}_y$	custom variable costs
k_{min}	(*optional*) maximum number of clusters
k_{max}	(*optional*) minimum number of clusters

The optimization type o_type states whether the objective value should be maximized or minimized. In the following we will only consider the maximization option because any minimization problem may be transformed into an equivalent maximization problem by negating the objective function. The clustering type c_type determines the kind of clustering to compute and is one of the three options *packing*, *partitioning* and *covering* described in Section 2.6.1. $G = (V, E)$ is a simple finite graph defining the nodes to be clustered and providing the connectivity information. The optional input values k_{min} and k_{max} provide an upper and lower bound for the number of clusters. If no bounds are given, it is always possible to define $k_{min} = 0$ and $k_{max} = |P(V)| = 2^n$.

Assume for now that a bound k_{max} is given and that $k_{min} = 0$. We define $K = [k_{max}]$ as the set of cluster indices and try to determine a node clustering $\mathcal{C} = \{C_1, \dots, C_{k_{max}}\} \subseteq P(V)$. Similar to Section 3.3, binary variables are applied to model which nodes belong to a cluster. However, for the CVCP we are dealing with multiple clusters so that each assignment variable x_{ik} has an additional cluster index k:

$$x_{ik} \in \{0,1\} \text{ with } x_{ik} = \begin{cases} 1 & \text{if } v_i \in C_k \\ 0 & \text{otherwise} \end{cases} \qquad \forall v_i \in V, \forall k \in K$$

We denote the vector of all assignment variables of cluster k as $\boldsymbol{x}_k = (x_{1k}, \dots, x_{nk})^T \in \{0,1\}^n$.

The CVCP can then be described as a MIP of the following form:

$$\max \quad \sum_{k \in K} \boldsymbol{c}_x^T \boldsymbol{x}_k + \boldsymbol{c}_y^T \boldsymbol{y}_k \tag{4.2}$$

$$\text{s.t.} \quad \sum_{k \in K} \boldsymbol{x}_k \begin{cases} \leq \mathbb{1} & \text{if } c\_type = packing \\ = \mathbb{1} & \text{if } c\_type = partitioning \\ \geq \mathbb{1} & \text{if } c\_type = covering \end{cases} \tag{4.3}$$

$$A_x \boldsymbol{x}_k + A_y \boldsymbol{y}_k \leq \boldsymbol{b} \qquad \forall k \in K \tag{4.4}$$

$$A'_x \boldsymbol{x}_k + A'_y \boldsymbol{y}_k \leq \boldsymbol{b'} \qquad \forall k \in K \tag{4.5}$$

$$\boldsymbol{x}_k \in \{0,1\}^n \qquad \forall k \in K \tag{4.6}$$

$$\boldsymbol{y}_k \in \mathcal{Y} \qquad \forall k \in K \tag{4.7}$$

Each column $\boldsymbol{y}_k \in \mathbb{R}^\ell$ of matrix $Y \in \mathbb{R}^{\ell \times |K|}$ is a vector of additional custom variables which may be used to define the cost and further constraints for cluster C_k. Vector $\boldsymbol{c}_x \in \mathbb{R}^n$ defines the costs of assigning a node to a cluster and $\boldsymbol{c}_y \in \mathbb{R}^\ell$ the costs of the custom variables. The objective function (4.2) is formed by the sum of these costs. The clustering constraints (4.3) vary depending on the type of clustering to compute.

The inequalities (4.5) represent the clusters' connectivity constraints. Section 3.3 showed that some approaches require more variables than the assignment variables x_{ik} alone for modeling connectivity. Such additional variables are considered to be part of the custom variables Y.

Matrix $A \in \mathbb{R}^{m \times (n+\ell)}$ and vector $\boldsymbol{b} \in \mathbb{R}^m$ define arbitrary additional linear constraints for each cluster on the variables $\boldsymbol{x}_k$ and $\boldsymbol{y}_k$. We divide A into $A_x \in \mathbb{R}^{m \times n}$ for the assignment variables and $A_y \in \mathbb{R}^{m \times \ell}$ for the custom variables. The custom constraints are then given by the inequalities (4.4).

Constraint (4.6) enforces binary values for the assignment variables. Input $\mathcal{Y} \subseteq \mathbb{R}^\ell$ determines the integer constraints on $\boldsymbol{y}_k$ via $\boldsymbol{y}_k \in \mathcal{Y}$.

Note that the actual number of clusters in an optimal solution may be smaller than the bound k_{max}. Therefore, feasible solutions must be allowed to contain one or more empty clusters which however should not to affect the objective function. For an empty cluster $C_k = \emptyset$, the total cost of all cluster variables $\boldsymbol{x}_k$ and $\boldsymbol{y}_k$ is thus assumed to amount to 0.

Furthermore, for typical CVCP instances it is possible to determine a far smaller value than 2^n for k_{max}. If *c_type* is *partitioning* or *packing*, then no node may appear in more than one cluster. Consequently, any feasible solution can comprise at most n non-empty clusters consisting of a single node each so that we can define $k_{max} = n$. When computing a covering (or overlapping clustering) this bound does not hold. In general, any connected cluster may be feasible. For fully connected graphs, all 2^n subgraphs are connected. However, for $k_{max} = 2^n$ already very small graphs will render the problem infeasible for standard solvers due to the exponential number of variables and corresponding constraints. Still, this is typically not an issue in practice since for most covering problems all non-empty clusters will affect the objective function negatively, i.e.,

$$\boldsymbol{c}_x^T \boldsymbol{x}_k + \boldsymbol{c}_y^T \boldsymbol{y}_k < 0 \quad \forall k \in K, C_k \neq \emptyset \quad . \tag{4.8}$$

Thus, each non-empty cluster of an optimal solution will cover at least one node that is not part of any other cluster. Again, we may define $k_{max} = n$. Moreover, in Section 4.3 we will remodel the problem in such a way that this issue becomes obsolete.

Note also that we assume all non-empty clusters of the solution to differ by at least one node. For packings and partitionings, this must hold because nodes may not appear in more than one cluster. For coverings this is not strictly enforced by the model, but again in practice typically satisfied by an optimal solution due to negative cluster costs for non-empty clusters. Consequently, multiple occurrences of the same cluster can be excluded. Nevertheless, it is also possible to demand unique clusters explicitly based on the boolean XOR operation and additional variables $z_{ik_1k_2}$:

$$\begin{aligned}
z_{ik_1k_2} &\leq x_{ik_1} + x_{ik_2} && \forall v_i \in V, \forall k_1 < k_2 \in K && (4.9)\\
z_{ik_1k_2} &\geq x_{ik_1} - x_{ik_2} && \forall v_i \in V, \forall k_1 < k_2 \in K && (4.10)\\
z_{ik_1k_2} &\geq x_{ik_2} - x_{ik_1} && \forall v_i \in V, \forall k_1 < k_2 \in K && (4.11)\\
z_{ik_1k_2} &\leq 2 - x_{ik_1} - x_{ik_2} && \forall v_i \in V, \forall k_1 < k_2 \in K && (4.12)\\
z_{ik_1k_2} &\in 0, 1 && \forall v_i \in V, \forall k_1 < k_2 \in K && (4.13)\\
\sum_{i=1}^{n} z_{ik_1k_2} &\geq \frac{1}{n} \sum_{i=1}^{n} x_{ik_1} && \forall k_1 < k_2 \in K && (4.14)
\end{aligned}$$

Constraints (4.9) – (4.13) ensure that $z_{ik_1k_2} = x_{ik_1} \oplus x_{ik_2}$. Hence, inequality (4.14) guarantees that any two clusters k_1 and k_2 differ by at least one node when cluster k_1 is non-empty.

Finally, the presented model does not yet take into account the lower bound k_{min} for the number of clusters. For $k_{min} > 0$, minor adaptions to the model are required. First, we add new binary variables z_k^{ne} to determine all non-empty clusters:

$$z_k^{ne} \in \{0, 1\} \text{ with } z_k^{ne} = \begin{cases} 1 & \text{if } C_k \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad \forall k \in K$$

Additional constraints ensure that all variables z_k^{ne} assume the intended value:

$$z_k^{ne} \leq \sum_{i=1}^{n} x_{ik} \qquad \forall k \in K$$
$$z_k^{ne} \geq \sum_{i=1}^{n} \frac{1}{n} x_{ik} \quad \forall k \in K$$

Then we require all feasible solutions to contain at least k_{min} non-empty clusters:

$$\sum_{k \in K} z_k^{ne} \geq k_{min}$$

In the following, we will not consider uniqueness constraints nor constraints for enforcing a minimum number of clusters to be part of the model. However, after remodeling the problem we arrive at a new formulation that prevents multiple occurrences of the same cluster and allows for both lower and upper bounds on the number of clusters.

4.2. Hardness

The CVCP is NP-hard because, e.g., the NP-hard Set Partitioning Problem from Section 2.6 can be reduced to it in polynomial time. For such a reduction, we define the graph of the CVCP to be complete with one node for each element of the universe S. Consequently, any subgraph is feasible based on the connectivity constraints alone. In order to ensure that only clusters corresponding to candidate subsets are feasible, we add binary custom variables $y_k^S \in \{0, 1\}$ for all $S \in (\mathcal{S} \cup \{\emptyset\})$ and the following custom constraints:

$$n \cdot y_k^S \leq \sum_{s_i \in S} x_{ik} + \sum_{s_i \notin S} 1 - x_{ik} \quad \forall S \in (\mathcal{S} \cup \{\emptyset\}), \forall k \in K \tag{4.15}$$
$$\sum_{S \in (\mathcal{S} \cup \{\emptyset\})} y_k^S = 1 \quad \forall k \in K \tag{4.16}$$

The constraints (4.15) enforce that y_k^S equals 1 iff cluster C_k corresponds to the subset S. Due to equation (4.16), each cluster C_k must thus be empty or correspond to a candidate subset. We select $c\_type = partitioning$ to obtain a partitioning and determine its cardinality by defining the objective function as the number of non-empty clusters:

$$\sum_{k \in K} z_k^{ne}$$

Let $o\_type = min$. For an optimal CVCP solution, the corresponding clustering is then a solution of the Set Packing Problem, i.e., a partitioning $\mathcal{C}^* \subseteq \mathcal{S}$ of minimal cardinality. Hence, the CVCP is NP-hard and no polynomial time algorithm solves this problem to optimality under the assumption that P $\neq$ NP.

4.3. Aggregated Extended Formulation

The previous section introduced what is called the *original formulation* of the CVCP. However, it is not ideal to solve the problem in the presented form. First of all, guaranteeing cluster connectivity as described in, e.g., Sections 3.3.1 or 3.3.2 requires a vast number of constraints. Fischetti et al. show that this problem may be solved by employing a branch-and-cut method [25]. Alternatively, a more efficient or heuristic approach for modeling connectivity might be found. As indicated in Section 4.1, another problem is that the number of cluster indices may grow exponentially in the size of the graph for some covering scenarios.

However, the main issue is less obvious and caused by the choice of the assignment variables x_{ik}. A node v_i belongs to cluster C_k iff $x_{ik} = 1$. Let us now consider an assignment comprising two clusters C_1 and C_2. Then we can obtain a new equivalent solution by simply exchanging clusters C_1 and C_2, i.e., we redefine $x_{i1}^{new} = x_{i2}$ and $x_{i2}^{new} = x_{i1}$. Hence, there are different representations for the exact same clustering. Generally, for any given solution of the original CVCP, any permutation of the cluster order leads to an equivalent assignment. This phenomenon of certain MIPs is known as *symmetry* and a common characteristic of related packing, covering and assignment problems [29, 44]. Unnecessarily having to examine equivalent solutions of a symmetric MIP may render the solver very inefficient.

Another consequence of the symmetry is that the LP solutions of the problem's linear relaxation may not provide information for deriving a clustering. For a feasible primal LP solution, let $\overline{\boldsymbol{x}}_k$ be the average of the assignment vectors, i.e.,

$$\overline{\boldsymbol{x}}_k = \frac{1}{k_{max}} \sum_{k \in K} \boldsymbol{x}_k \quad .$$

If all assignment vectors are defined as $\overline{\boldsymbol{x}}_k$, we obtain a new solution where the clustering constraints are still satisfied but each node is equally assigned to all clusters. Note that the new solution has the same objective value as the original one. Consequently, even an optimal solution may not group any nodes into clusters.

In order to avoid the aforementioned issues, we transform the CVCP by applying the so-called *Dantzig-Wolfe decomposition* [16, 17]. This decomposition divides the original problem into a master problem and one or more subproblems. All problems are then solved in a combined iterative process that leads to an optimal solution of the original problem. There are different variants of Dantzig-Wolfe decompositions. The more general convexity approach expresses the original variables via a convex combination of extreme points and extreme rays [19, 18, 43, 53]. In our case, we will focus on an alternate discretization technique particularly targeting at MIPs [28, 43, 54, 56]. However, since only the binary variables $x_{ik} \in \{0, 1\}$ are transformed, both approaches in fact coincide for the CVCP [43].

Examining the original CVCP formulation from Section 4.1, it can be seen that many constraints only comprise variables related to one cluster. Thus, the CVCP has the following overall structure:

$$\begin{pmatrix} B_1 & B_2 & \cdots & B_{|K|} \\ D & & & \\ & D & & \\ & & \ddots & \\ & & & D \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_{|K|} \end{pmatrix} \leq \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{d} \\ \boldsymbol{d} \\ \vdots \\ \boldsymbol{d} \end{pmatrix} \tag{4.17}$$

Here, we define vector $\boldsymbol{v}_k^T = (\boldsymbol{x}_k^T, \boldsymbol{y}_k^T)$ as the vector comprising all variables that belong to cluster C_k. Matrix D incorporates the custom constraints (4.4) and the connectivity constraints (4.5). These are exactly the constraints restricted to occurrences of variables $\boldsymbol{v}_k$ of only a single cluster C_k. In contrast, the clustering constraints (4.3) incorporate variables of all $\boldsymbol{v}_k$ and are jointly defined by the matrices B_k.

Since the constraints defined by matrix D are limited to a certain set of variables, they provide the problem with a special structure. This becomes obvious through the characteristic diagonal blocks in the constraint matrix which imply that all variables have non-zero coefficients in at most one block. The constraints defined by these block matrices are therefore called *structural constraints*. The remaining constraints based on the matrices B_k are referred to as *linking constraints* because they bind variables from multiple blocks together.

Dantzig-Wolfe decomposition now exploits the special structure of the constraint matrix for the separation of the original problem into smaller problems that can be solved more efficiently. Let $\mathcal{X}$ be the set of all feasible $\boldsymbol{x}_k$-vectors based on the structural and integer constraints alone:

$$\mathcal{X} = \{\boldsymbol{x} \in \{0,1\}^n \mid \exists \boldsymbol{y} \in \mathcal{Y}\colon D \cdot (\boldsymbol{x}, \boldsymbol{y}) \leq \boldsymbol{d}\}$$

Let further $z_y^{max}(\boldsymbol{x})$ be the maximum value of $\boldsymbol{c}_y^T \boldsymbol{y}$ for a given $\boldsymbol{x} \in \mathcal{X}$:

$$z_y^{max}(\boldsymbol{x}) = \max\{\boldsymbol{c}_y^T \boldsymbol{y} \mid D \cdot (\boldsymbol{x}, \boldsymbol{y}) \leq \boldsymbol{d}\}$$

An alternative formulation of the original CVCP is then:

$$\begin{aligned} \max \quad & \sum_{k \in K} \left(\boldsymbol{c}_x^T \boldsymbol{x}_k + z_y^{max}(\boldsymbol{x}_k) \right) \\ \text{s.t.} \quad & \sum_{k \in K} \boldsymbol{x}_k \begin{cases} \leq \mathbb{1} & \text{if } c\_type = packing \\ = \mathbb{1} & \text{if } c\_type = partitioning \\ \geq \mathbb{1} & \text{if } c\_type = covering \end{cases} \\ & \boldsymbol{x}_k \in \mathcal{X} \qquad\qquad\qquad\qquad \forall k \in K \end{aligned}$$

Note that the new model solely depends on the assignment variables x_{ik}. Each vector $\boldsymbol{x} \in \mathcal{X}$ defines a unique feasible cluster $C \subseteq V$ based on the node assignments, i.e., $C = \{v_i \mid i \in supp(\boldsymbol{x})\}$. Define $\mathcal{V} = \{V_1, \ldots, V_{n_v}\}$ as the set of all feasible clusters with $n_v = |\mathcal{V}|$. Note that the mapping

$$map \quad : \quad \mathcal{X} \to \mathcal{V}, \quad \boldsymbol{x} \mapsto C = \{v_i \mid i \in supp(\boldsymbol{x})\}$$

is bijective since the inverse function is given by the characteristic vector $\boldsymbol{x}_C$ of cluster C. In order to avoid the redundant solutions caused by symmetry, we define a new representation of the solution space:

$$\begin{aligned}\mathcal{X} &= \left\{ \sum_{C \in \mathcal{V}} \lambda_C \cdot map^{-1}(C) \mid \sum_{C \in \mathcal{V}} \lambda_C = 1, \boldsymbol{\lambda} \in \{0,1\}^{n_v} \right\} \\ &= \left\{ \sum_{C \in \mathcal{V}} \lambda_C \cdot \boldsymbol{x}_C \mid \sum_{C \in \mathcal{V}} \lambda_C = 1, \boldsymbol{\lambda} \in \{0,1\}^{n_v} \right\} \qquad (4.18)\end{aligned}$$

Essentially, we represent any vector $\boldsymbol{x} \in \mathcal{X}$ in terms of a vector $\boldsymbol{\lambda}$ by setting the entry λ_C of the corresponding cluster C to 1. Thus, we introduce new cluster variables λ_{Ck} to substitute for $\boldsymbol{x}_k$:

$$\lambda_{Ck} \in \{0,1\} \text{ with } \lambda_{Ck} = \begin{cases} 1 & \text{if solution cluster } C_k \text{ equals} \\ & \text{feasible cluster } C \\ 0 & \text{otherwise} \end{cases} \qquad \forall C \in \mathcal{V}, \forall k \in K$$

Let $\boldsymbol{\lambda}_k = (\lambda_{V_1 k}, \ldots, \lambda_{V_{n_v} k})^T$. The corresponding cost vector $\boldsymbol{c}_\lambda$ is derived from the total cluster costs:

$$(\boldsymbol{c}_\lambda)_C = \boldsymbol{c}_x^T \boldsymbol{x}_C + z_y^{max}(\boldsymbol{x}_C) \quad \forall C \in \mathcal{V} \qquad (4.19)$$

Substituting for $\boldsymbol{x}_k$ as in equation (4.18) results in an *extended formulation* of the CVCP:

$$\begin{aligned} \max \quad & \sum_{k \in K} \boldsymbol{c}_\lambda \boldsymbol{\lambda}_k \\ \text{s.t.} \quad & \sum_{k \in K} \sum_{C \in \mathcal{V}} \lambda_{Ck} \cdot \boldsymbol{x}_C \begin{cases} \leq \mathbb{1} & \text{if } c\_type = packing \\ = \mathbb{1} & \text{if } c\_type = partitioning \\ \geq \mathbb{1} & \text{if } c\_type = covering \end{cases} \\ & \sum_{C \in \mathcal{V}} \lambda_{Ck} = 1 & \forall k \in K \qquad (4.20) \\ & \boldsymbol{\lambda}_k \in \{0,1\}^{n_v} & \forall k \in K \end{aligned}$$

Nevertheless, the symmetry issue of redundant solutions still remains. Arbitrary permutations of the vectors $\boldsymbol{\lambda_k}$ are different representations of the same clustering. However, with a small adaptation we can prevent this problem and even reduce the number of variables and hence the complexity of the model. Therefore, we first declare the empty cluster infeasible by excluding it from $\mathcal{V}$ and then aggregate variables over the clusters. Since all clusters in the solution are supposed to be different, the selected clusters can be encoded by a single vector $\boldsymbol{\lambda} \in \{0,1\}^{n_v}$:

$$\lambda_C \in \{0,1\} \text{ with } \lambda_C = \begin{cases} 1 & \text{if feasible cluster } C \text{ forms} \\ & \text{part of the solution } \mathcal{C} \\ 0 & \text{otherwise} \end{cases} \qquad \forall C \in \mathcal{V}$$

Note, that the cardinality 2^{n_v} of the power set $P(\mathcal{V})$ corresponds to the number of possible values for $\boldsymbol{\lambda}$. Hence, the mapping from $\boldsymbol{\lambda}$ to the feasible clusterings is bijective.

We avoid symmetry and obtain the *aggregated extended formulation* of the CVCP:

$$\begin{aligned} \max \quad & \boldsymbol{c}_\lambda^T \boldsymbol{\lambda} \\ \text{s.t.} \quad & \sum_{C \in \mathcal{V}} \lambda_C \cdot \boldsymbol{x}_C \begin{cases} \leq \mathbb{1} & \text{if } c\_type = packing \\ = \mathbb{1} & \text{if } c\_type = partitioning \\ \geq \mathbb{1} & \text{if } c\_type = covering \end{cases} \\ & \boldsymbol{\lambda} \in \{0,1\}^{n_v} \end{aligned} \tag{4.21}$$

The correctness of the model follows almost directly from the extended formulation by replacing any sum over the $\boldsymbol{\lambda}_k$ with just the single vector $\boldsymbol{\lambda}$. Additionally, equation (4.20) is dropped since $\boldsymbol{\lambda}$ does not just encode one but all selected clusters. Without any additional constraints, any cluster $C \in \mathcal{V}$ could so far in theory still have been selected by multiple $\boldsymbol{\lambda}_k$. As mentioned in Section 4.1, this behavior is actually not desired. Thus, the aggregated extended formulation prevents multiple cluster occurrences without having to add any further constraints to the model. As each entry of value 1 in $\boldsymbol{\lambda}$ corresponds to a selected non-empty cluster, the definition of bounds for the number of clusters is also simple and straight forward:

$$\begin{aligned} \sum_{C \in \mathcal{V}} \lambda_C &\geq k_{min} \\ \sum_{C \in \mathcal{V}} \lambda_C &\leq k_{max} \end{aligned} \tag{4.22}$$

However, even with the aggregated extended formulation there still is one major issue. The number of subsets of V and thus the amount of potentially feasible clusters grows exponentially in n. Hence, solving the problem with a standard solver is not an option for larger instances. Consequently, we solve the problem based on the new model via a branch-and-price approach that is described in the following.

5. Method

As already mentioned, the aggregated extended formulation still comprises a large number of variables. Also, the transformed model requires restricting the cluster variables λ_j to feasible clusters only. Hence, employing a standard solver additionally requires the precomputation of all feasible clusters and their corresponding costs before the algorithm can even start. Instead, we opt for a branch-and-price approach [5, 18, 42, 43]. This method is particularly useful for problems with a vast number of variables but only few constraints, such as the aggregated extended CVCP. Figure 5.1 illustrates the concept's general outline.

The main idea is to reduce the size of the original problem by considering only a

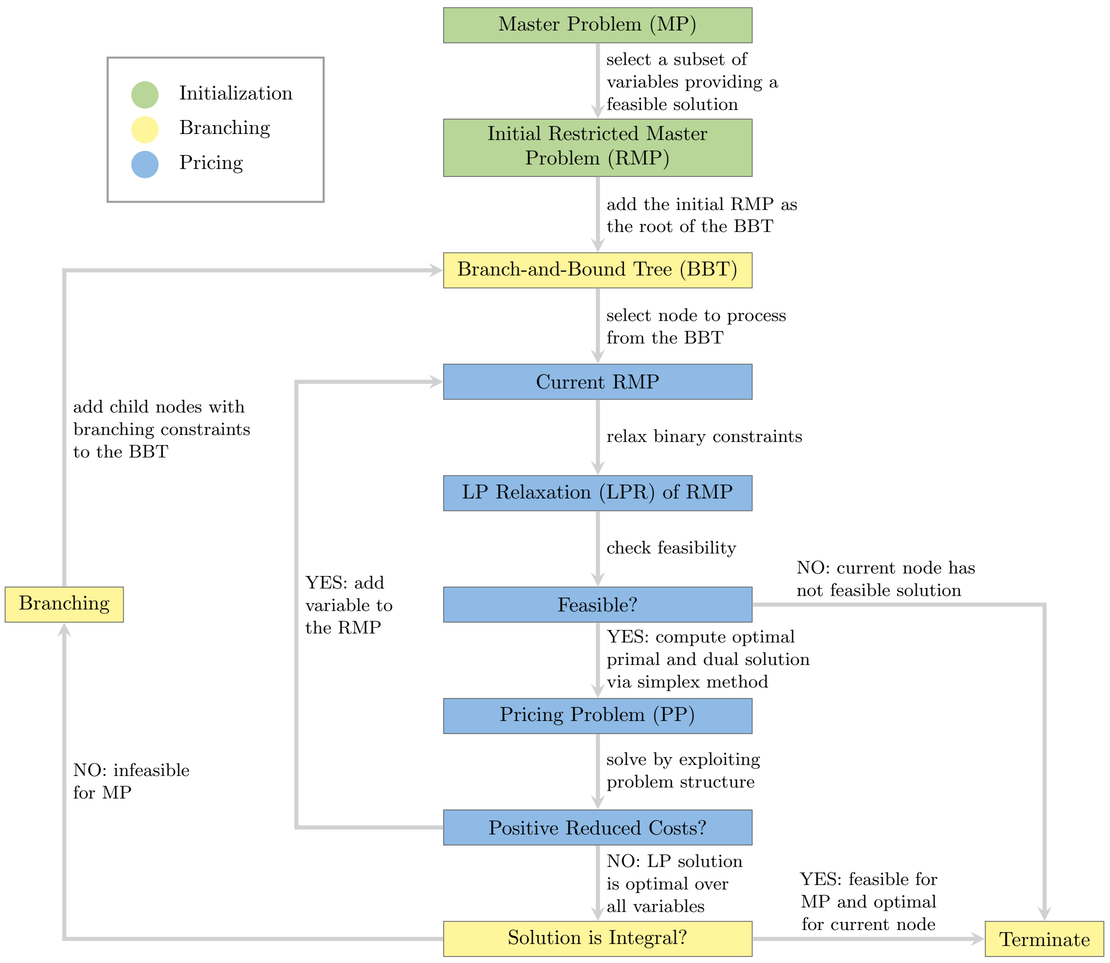

Initialization
Branching
Pricing
Master Problem (MP)
select a subset of variables providing a feasible solution
Initial Restricted Master Problem (RMP)
add the initial RMP as the root of the BBT
Branch-and-Bound Tree (BBT)
select node to process from the BBT
Current RMP
relax binary constraints
LP Relaxation (LPR) of RMP
check feasibility
Feasible?
NO: current node has not feasible solution
YES: compute optimal primal and dual solution via simplex method
Pricing Problem (PP)
solve by exploiting problem structure
Positive Reduced Costs?
YES: add variable to the RMP
NO: LP solution is optimal over all variables
Solution is Integral?
YES: feasible for MP and optimal for current node
Terminate
NO: infeasible for MP
Branching
add child nodes with branching constraints to the BBT

Figure 5.1.: *General outline of the concept of branch-and-price, given that the MP is feasible. The approach can be divided into three main steps which are represented by different colors.*

limited number of variables. In the initialization phase, a small subset of variables is selected for which the problem is still feasible. These can, e.g., be the base variables of a known feasible solution. The branch-and-bound method described in Section 2.5 is then applied to the smaller problem in order to compute an integer solution. Additionally, this method is extended by the so-called *pricing loop*. Each time the LPR of the small problem is solved to optimality at some node in the branch-and-bound tree, it is checked whether the introduction of any missing variables may improve the solution. If so, a suitable variable is added to the small problem. Since introducing a new variable is equivalent to adding a new column to the simplex tableau, pricing is also called *column generation*. When no improving variables are left, the pricing loop ends and standard branch-and-bound continues. This way, branching and pricing alternates until all branches of the branch-and-bound tree have terminated. Carried out correctly, branch-and-price is an exact solution approach.

The remainder of this chapter explains our method and the general concept of branch-and-price in more detail. First, the master problem and the restricted master problem are formally introduced in Section 5.1. Afterwards, Sections 5.2 and 5.3 describe how the pricing and the branching are performed. Finally, Section 5.4 provides a summary of the whole approach and examines some additional aspects.

5.1. Master Problem

Branch-and-price is applied with the goal of finding an optimal solution to a MIP which in this case is the aggregated extended formulation of the CVCP. In the framework of branch-and-price, we refer to this model as the *master problem* (MP). Assume for simplicity that the MP is of the form

$$\begin{aligned} \max \quad & \sum_{C \in \mathcal{V}} c_C \lambda_C \\ \text{s.t.} \quad & \sum_{C \in \mathcal{V}} \boldsymbol{a}_C \lambda_C \leq \boldsymbol{b} \\ & \lambda_C \in \{0,1\} \quad \forall C \in \mathcal{V} \end{aligned}$$

where matrix $A \in \mathbb{R}^{m \times n_c}$ and vector $\boldsymbol{b}$ are the left and right side of the linear constraints. Vector $\boldsymbol{a}_C$ is the column of A corresponding to variable λ_C.

Originating from the MP, a subset of the variables is chosen to start out with. Assume, e.g., that a heuristic method is employed to determine a feasible solution $\boldsymbol{\lambda}_{init}$ of the MP. The selected variables can then be defined as the cluster variables of the corresponding set of clusters $\mathcal{V}' \subseteq \mathcal{V}$ defined as

$$\mathcal{V}' = \{V_i \in \mathcal{V} \mid i \in supp(\boldsymbol{\lambda}_{init})\} \quad .$$

A *restricted master problem* (RMP) is derived as the problem with the same objective function and constraints as the MP but restricted to the selected variables:

$$\begin{aligned} \max \quad & \sum_{C \in \mathcal{V}'} c_C \lambda_C \\ \text{s.t.} \quad & \sum_{C \in \mathcal{V}'} \boldsymbol{a}_C \lambda_C \leq \boldsymbol{b} \\ & \lambda_C \in \{0,1\} \quad \forall C \in \mathcal{V}' \end{aligned} \tag{5.1}$$

Replacing the binary constraints (5.1) with the inequalities

$$0 \leq \lambda_C \leq 1 \quad \forall C \in \mathcal{V}' \tag{5.2}$$

yields the corresponding LPR. In the partitioning and packing scenario, the upper bounds on the cluster variables are obsolete because they are indirectly enforced by the clustering constraints from equation (4.21). As long as no covering is computed, these bounds are therefore removed from the problem. This does not just reduce the size of the model but also the amount of resulting dual variables. The advantage of omitting unrequired dual variables becomes apparent later in Sections 5.2 and 5.3.

In order to solve the LPR with the simplex method, it is transformed into *slack form*. Therefore, all inequalities are replaced by equations. Additionally, a suitable *slack variable* is introduced for each former inequality to allow the equation to be satisfied:

$$\begin{aligned} \max \quad & c_s^T \boldsymbol{\lambda}_s \\ \text{s.t.} \quad & A_s \boldsymbol{\lambda}_s = \boldsymbol{b}_s \\ & \boldsymbol{\lambda}_s \geq 0 \end{aligned}$$

$\boldsymbol{\lambda}_s$ is the vector including all variables of the LPR and the new slack variables. Matrix A_s and vector $\boldsymbol{b}_s$ are the left and right side of the constraints, including also the upper bounds from inequalities (5.2). The application of the simplex method renders an optimal solution $\boldsymbol{\lambda}_s^*$ for this slack form representation of the RMP's LPR.

5.2. Pricing

The RMP was deduced from the MP by dropping many of the original variables. Then, we applied the simplex method to solve the slack form LPR of the RMP. In order to find an optimal solution for the original MP, we now search for variables which are not yet part of the RMP and help to improve the objective function value of the current LPR.

5.2.1. Reduced Costs

For this purpose, we split the slack form variables $\boldsymbol{\lambda}_s$ into the basic variables $\boldsymbol{\lambda}_B$ and the non-basic variables $\boldsymbol{\lambda}_N$ and A_s into A_B and A_N accordingly. Note that A_B is an invertible square matrix as it corresponds to the basic variables, whose columns are linearly independent. It holds:

$$A_s \boldsymbol{\lambda}_s = \boldsymbol{b}_s \Leftrightarrow A_B \boldsymbol{\lambda}_B + A_N \boldsymbol{\lambda}_N = \boldsymbol{b}_s \Leftrightarrow \boldsymbol{\lambda}_B = A_B^{-1} \boldsymbol{b}_s - A_B^{-1} A_N \boldsymbol{\lambda}_N$$

We insert this equation into objective function:

$$\begin{aligned} \boldsymbol{c}_s^T \boldsymbol{\lambda}_s = \boldsymbol{c}_B^T \boldsymbol{\lambda}_B + \boldsymbol{c}_N^T \boldsymbol{\lambda}_N &= \boldsymbol{c}_B^T \Big(A_B^{-1} \boldsymbol{b}_s - A_B^{-1} A_N \boldsymbol{\lambda}_N \Big) + \boldsymbol{c}_N^T \boldsymbol{\lambda}_N \\ &= \boldsymbol{c}_B^T A_B^{-1} \boldsymbol{b}_s + \Big(\underbrace{\boldsymbol{c}_N^T - \boldsymbol{c}_B^T A_B^{-1} A_N}_{\bar{\boldsymbol{c}}_N^T} \Big) \boldsymbol{\lambda}_N \end{aligned} \tag{5.3}$$

Vector $\bar{\boldsymbol{c}}_N^T$ is the so-called reduced cost indicating how a change of the non-basic variables affects the objective function value. If the MP has a better solution than the optimal solution $\boldsymbol{\lambda}_s^*$ of the LPR, then there must be some variable λ_{C^*} with $C^* \notin \mathcal{V}'$ whose introduction results in an increase of the linear objective function. In analogy to the pricing step of the simplex method, it is checked if a suitable λ_{C^*} exists. As λ_{C^*} is yet unintroduced, its current value is zero. Hence, we may add it to the tableau without impacting any other variables or constraints and consider it as a non-basic variable. Due to $\lambda_{C^*} \geq 0$ and equation (5.3), it is necessary that λ_{C^*} has positive reduced costs:

$$\begin{aligned} 0 &< (\bar{c}_N)_{C^*} \qquad (5.4) \\ &= (c_N)_{C^*} - (c_B^T A_B^{-1} A_N)_{C^*} \\ &= (c_N)_{C^*} - (\boldsymbol{\pi}_s^*)^T (\boldsymbol{a}_N)_{C^*} \end{aligned}$$

Here, $\boldsymbol{\pi}_s^*$ is the optimal dual solution corresponding to $\boldsymbol{\lambda}_s^*$.

5.2.2. Pricing Problem

In order to identify a suitable cluster $C^* \in \mathcal{V} \setminus \mathcal{V}'$ that satisfies inequality (5.4), we define the following subproblem called *pricing problem (PP)*:

$$C^* = \arg\max_{C \in \mathcal{V}} \{ (c_N)_C - (\boldsymbol{\pi}_s^*)^T (\boldsymbol{a}_N)_C \} \tag{5.5}$$

Note that all feasible clusters $C \in \mathcal{V}$ are being considered here, including the ones in $\mathcal{V}'$ that do already form part of the RMP. However, for an optimal slack form solution $\boldsymbol{\lambda}_s^*$ and the corresponding dual solution $\boldsymbol{\pi}_s^*$, it can be shown that

$$0 \geq (c_N)_C - (\boldsymbol{\pi}_s^*)^T (\boldsymbol{a}_N)_C \quad \forall C \in \mathcal{V}' \quad .$$

Hence, in the case of $C^* \in \mathcal{V}'$ there is no feasible cluster fulfilling condition (5.4).

At first, solving the PP seems to render the entire solution approach useless. It requires maximization over all feasible clusters $C \in \mathcal{V}$ which is exactly what we tried to omit all along. However, exploiting the problem structure allows to solve this problem more efficiently. Instead of explicit optimization over all clusters, the best cluster is computed directly by formulating the PP as the following MIP:

$$\begin{aligned} \max \quad & (\boldsymbol{c}_x - \boldsymbol{\pi}_x^*)^T \boldsymbol{x} + \boldsymbol{c}_y^T \boldsymbol{y} \\ \text{s.t.} \quad & A_x \boldsymbol{x} + A_y \boldsymbol{y} \leq \boldsymbol{b} \\ & A_x' \boldsymbol{x} + A_y' \boldsymbol{y} \leq \boldsymbol{b}' \\ & \boldsymbol{x} \in \{0,1\}^n \\ & \boldsymbol{y} \in \mathcal{Y} \end{aligned}$$

This formulation is similar to the original CVCP in Section 4.1 and we use the same definitions. The most significant difference is that only a single cluster is being considered here. This means in particular that the previous symmetry issues no longer apply. The variable index k of the variables $\boldsymbol{x}$ and $\boldsymbol{y}$ was dropped. Moreover, we define $(\boldsymbol{\pi}_x^*)_i$ as the components of $\boldsymbol{\pi}_s^*$ which corresponds to the clustering constraint of node v_i.

The goal is now to show that the new MIP corresponds to the PP (5.5). For any feasible solution, the equivalent cluster is derived as in Chapter 4.3, i.e., $C = \{v_i \mid i \in supp(\boldsymbol{x})\ \}$. The same chapter illustrates that the feasible solutions of the PP correspond exactly to the feasible clusters of the MP. According to the definition of the MP's cluster costs (4.19), it further holds:

$$(c_N)_C \geq \boldsymbol{c}_x^T \boldsymbol{x} + \boldsymbol{c}_y^T \boldsymbol{y}$$

Equality is met iff $\boldsymbol{y}$ maximizes the objective function for the given $\boldsymbol{x}$. In particular, this is also the case for any optimal solution $(\boldsymbol{x}^*, \boldsymbol{y}^*)$. Hence, only the term $(\boldsymbol{\pi}_s^*)^T(\boldsymbol{a}_N)_C$ of the PP definition remains to be matched.

For any constraint in the MP that does not contain variable λ_C, it holds $((\boldsymbol{a}_N)_C)_i = 0$ for the corresponding row i. Consider the clustering constraints (4.21) first. Variable λ_C occurs precisely in the clustering constraints of the nodes $v_t \in C$. It follows:

$$(\boldsymbol{\pi}_s^*)_i((\boldsymbol{a}_N)_C)_i = (\boldsymbol{\pi}_x^*)_t x_t$$

The second group of constraints are the variables' upper bounds (5.2). We distinguish between the three clustering types.

Partitioning and Packing

As explained in Section 5.1, the bounds are removed for the partitioning and packing scenario. Hence, the MIP is equivalent to the PP (5.5).

Covering

In the covering scenario, the upper bounds are present. For any row i corresponding to an upper bound, it holds $((\boldsymbol{a}_N)_C)_i = 1$ iff λ_C is the bounded variable. Hence, the corresponding dual value $(\boldsymbol{\pi}_s^*)_i$ only becomes relevant if the computed cluster C is already part of the RMP. However, in this case its reduced costs are non-positive. Even if C was the optimal solution of the PP, it would not pass condition (5.4) and would not be added to the RMP. Thus, we do not consider the term $(\boldsymbol{\pi}_s^*)_i((\boldsymbol{a}_N)_C)_i$ in the MIP and accept that the objective function value may not correspond to the reduced costs for any RMP cluster in $\mathcal{V}'$. Instead, we add constraints to ensure that these clusters are declared infeasible if their objective function value is positive. The following constraint renders a single cluster C infeasible:

$$\sum_{v_i \in C} (1 - x_i) + \sum_{v_i \notin C} x_i \geq 1 \tag{5.6}$$

Depending on whether a lower bound $k_{min} > 0$ for the number of clusters is given or not, we add the exclusion constraint (5.6) for different variables.

First, let no lower bound $k_{min} > 0$ be given. Hypothetically, we assume that there is a feasible MP solution containing multiple occurrences of the same cluster. Note that removing a duplicate cluster from a solution cannot violate any covering constraint, branching constraint, upper bound k_{max} for the number of clusters or variable bound. Hence, after removing all duplicates the given solution remains feasible. Let now $C \in \mathcal{V}'$ be a cluster present in the RMP. If cluster C has non-positive costs $c_C \leq 0$, then multiple occurrences cannot improve the objective function. It holds $(\boldsymbol{\pi}_s^*)_i = 0$ and the MIP computes the reduced costs of C correctly. Hence, there is no need to declare cluster C infeasible. However, $(\boldsymbol{\pi}_s^*)_i = 0$ does not hold if cluster C has cost $c_C > 0$. In this case, the MIP computes the reduced costs of cluster C incorrectly and we add a corresponding exclusion constraint.

Assume now that a lower bound $k_{min} > 0$ is given. If a feasible solution would contain multiple occurrences of the same cluster, the removal of the duplicates might now violate the lower bound. Consequently, for any cluster $C \in \mathcal{V}'$ it may hold$(\boldsymbol{\pi}_s^*)_i \neq 0$, even if its cost is non-positive, i.e., $c_C \leq 0$. Therefore, exclusion constraints are added for all clusters in the RMP.

5.2.3. Pricing Loop

Based on the PP, we can now compute an optimal cluster $C^* \in \mathcal{V}$ to improve the objective function value. If C^* satisfies inequality (5.4), it further holds $C^* \notin \mathcal{V}'$ and the corresponding cluster variable λ_{C^*} is inserted into the RMP. Then, both the LPR of the RMP and the PP are resolved. This pricing loop continues until the reduced costs of λ_{C^*} are non-positive. In this case, the solution $\boldsymbol{\lambda}^*$ is not just optimal for the LPR of the RMP but also for the LPR of the MP itself. If $\boldsymbol{\lambda}^*$ is integral, it is also feasible for the MP and the current branch terminates. Otherwise, $\boldsymbol{\lambda}^*$ does not satisfy the integer constraints and is therefore infeasible for the MP. According to branch-and-bound, we continue by applying a branching rule to split the current RMP into smaller subproblems. The details of branching and its effects on the PP are described in Section 5.3.

5.3. Branching

Assume that an optimal solution $\boldsymbol{\lambda}^*$ for the current LPR was computed and no further improving variables can be introduced. If the solution is fractional, it is infeasible for the MP and we compute an integral one via a standard branch-and-bound strategy as described in Section 2.5.

5.3.1. Branching in the Master Problem

Section 2.5 introduced variable branching as a branching rule. Applied to the CVCP, each of the two created subproblems would contain a new upper or lower bound for some variable with fractional solution λ_C^* as a branching constraint:

$$\lambda_C \leq \lfloor \lambda_C^* \rfloor$$
$$\lambda_C \geq \lceil \lambda_C^* \rceil$$

Section 3.2 described why this branching rule is not recommended for the CVCP and presented Ryan-Foster branching as an alternative for partitioning problems. In the following, we will recall this method in the context of the CVCP and then adapt it for the packing and covering scenario.

Partitioning

For the CVCP, the candidate subsets from Section 3.2 are given by the set of feasible clusters $C \in \mathcal{V}'$ in the RMP. Let $\boldsymbol{\lambda}^*$ be an optimal solution of the current node's LPR with fractional cluster variable λ_{C_a}, i.e., $0 < \lambda^*_{C_a} < 1$. We know that a node $v_s \in C_a$ exists which is also part of a second fraction cluster C_b. Moreover, all clusters are different and thus there exists some node $v_t \in (C_a \triangle C_b) \subseteq (V \setminus \{v_s\})$. Adapting the inequalities (3.1) to the CVCP, it holds:

$$0 < \sum_{C \in \mathcal{V}' : \{v_s, v_t\} \subseteq C} \lambda^*_C < 1 \tag{5.7}$$

The branching constraints for Ryan Forster branching are then given by the following two equations:

$$\sum_{C \in \mathcal{V}' : \{v_s, v_t\} \subseteq C} \lambda_C = 1 \tag{5.8}$$

$$\sum_{C \in \mathcal{V}' : \{v_s, v_t\} \subseteq C} \lambda_C = 0 \tag{5.9}$$

Constraint (5.8) belongs to the same-branch where the two nodes v_s and v_t must be part of the same cluster. For the differ-branch, constraint (5.9) ensures that the two nodes do not occur together in any cluster. Due to the inequalities (5.7), either constraint guarantees that the previous fractional optimal solution $\boldsymbol{\lambda}^*$ is no longer feasible in the corresponding subproblem.

Packing

We apply the same branching constraints (5.8) and (5.9) as for the partitioning case. Therefore, we first prove the following theorem:

Theorem 5.1. *Let $\boldsymbol{\lambda}^*$ be an optimal basic solution of the LPR at some node in the branch-and-bound tree of a CVCP with c_type = packing. Let the solution further contain a fractional cluster C_a. Then it holds:*

$$\exists v \in C_a : \sum_{C \in \mathcal{V}' : v \in C} \lambda^*_C = 1 \tag{5.10}$$

Proof. Due to $\boldsymbol{\lambda}^* \geq 0$ and $\lambda^*_{C_a} > 0$, the cluster variable λ_{C_a} does not occur in any differ-branching constraint (5.9). We distinguish three cases:

Case 1. Variable $\lambda^*_{C_a}$ forms part of a same-branching constraint (5.8) for two nodes $v_s, v_t \in C_a$. It holds :

$$1 = \sum_{C \in \mathcal{V}' : \{v_s, v_t\} \subseteq C} \lambda^*_C \leq \sum_{C \in \mathcal{V}' : v_s \in C} \lambda^*_C \leq 1$$

where the last inequality is given by the packing constraint (4.21) of node v_s. Since all terms must be equal, the theorem holds for $v = v_s$.

Case 2. Variable $\lambda^*_{C_a}$ does not occur in any same-branching constraint, but the packing constraint (4.21) of some node $v' \in C_a$ is satisfied with equality. Again, it follows

$$\sum_{C \in \mathcal{V}' : v' \in C} \lambda^*_C = 1$$

and the theorem holds for $v = v'$.

Case 3. Variable $\lambda^*_{C_a}$ does not occur in any same-branching constraint and there is no node $v' \in C_a$ whose packing constraint is satisfied with equality. We show that this leads to a contradiction. Let $\boldsymbol{\lambda}_B$ be the vector of basic variables. We denote the basic matrix by A_B and the corresponding right hand side by $\boldsymbol{b}_B$. It holds:

$$A_B^{-1} \boldsymbol{b}_B = \boldsymbol{\lambda}^*_B \tag{5.11}$$

Note that A_B contains only *tight* rows, i.e., equations corresponding to constraints that are satisfied with equality. Moreover, λ_{C_a} is greater than 0 and thus one of the basic variables in $\boldsymbol{\lambda}_B$. None of the packing constraints containing λ_{C_a} are tight and the variable does not occur in any branching constraints. The upper bound (5.2) of λ_{C_a} is not tight either because the variable is fractional. Thus, the only remaining constraints containing λ_{C_a} are the bounds on the number of clusters, if defined. We distinguish two subcases:

Case 3.1. The CVCP contains a lower or upper bound for the number of clusters which is satisfied with equality. If $k_{min} < k_{max}$, then only one of the two constraints is tight. Otherwise, $k_{min} = k_{max}$ and both bounds correspond to the same equation. Hence, one equation is redundant and not part of A_B. Either way, λ_{C_a} occurs only in a single row of A_B. W.l.o.g., let this row be the last. It holds $(\boldsymbol{a}_B)_{C_a} = (0, \ldots, 0, 1)^T$. All entries of A_B and $\boldsymbol{b}_B$ are integers. Moreover, the last row of A_B only contains the values 0 and 1. Hence, solving equation (5.11) for $\boldsymbol{\lambda}^*_B$ via Gaussian Elimination results in $\lambda^*_{C_a} \in \mathbb{Z}$. This contradicts the premise of $\lambda^*_{C_a}$ being fractional.

Case 3.2. The CVCP contains no bounds for the number of clusters or these bounds are not satisfied with equality. It follows $(\boldsymbol{a}_B)_{C_a} = \mathbb{0}$, which contradicts the premise that $\boldsymbol{\lambda}^*$ is a basic solution.

Since all subcases of Case 3 result in a contradiction, this case cannot occur and the theorem holds. □

Assume now that an optimal basic solution $\boldsymbol{\lambda}^*$ with a fractional cluster C_a is given. Theorem 5.1 states that there is some node $v_s \in C_a$ for which equation (5.10) holds. Since $0 < \lambda^*_{C_a} < 1$, the node v_s must belong to a second fractional cluster C_b. All clusters are pairwise different and hence there exists a node $v_t \in (C_a \triangle C_b) \subseteq (V \setminus \{v_s\})$. Only one of the two clusters C_a and C_b contains v_t and the inequalities (5.7) hold.

Thus, we branch based on the same constraints as for the partitioning case. Since each node occurs in at most one cluster, every feasible solution of the parent MIP will again satisfy either equation (5.8) or equation (5.9). Note that in contrast to the partitioning scenario, not every node $v_s \in C_a$ must reoccur in another cluster. To identify a suitable v_s, one may iterate over the nodes of the other fractional clusters until a shared node is found.

Covering

We extend Ryan-Foster branching to the covering scenario. Therefore, the branching constraints are defined as

$$\sum_{C \in \mathcal{V}' : V' \subseteq C} \lambda_C \geq 1 \tag{5.12}$$

$$\sum_{C \in \mathcal{V}' : V' \subseteq C} \lambda_C = 0 \tag{5.13}$$

for some node set $V' \subseteq V$ with $|V'| \geq 2$. For $|V'| = 2$, these branching constraints are similar to those of the partitioning and packing scenario. For consistency, we again refer to the branch based on constraint (5.12) as the same-branch and the branch derived from constraint (5.13) the differ-branch.

First we show the following theorem:

Theorem 5.2. *Let $\boldsymbol{\lambda}^*$ be an optimal basic solution of the LPR at some node in the branch-and-bound tree of a CVCP with c_type = covering. Let the solution further contain a fractional cluster C_a. Then it holds:*

$$\exists V' \subseteq C_a : \quad V' \neq \emptyset \quad \wedge \quad \sum_{C \in \mathcal{V}' : V' \subseteq C} \lambda^*_C = 1 \tag{5.14}$$

Proof. Due to $\boldsymbol{\lambda}^* \geq 0$ and $\lambda^*_{C_a} > 0$, the cluster variable λ_{C_a} does not occur in any differ-branching constraint (5.13). We distinguish two cases:

Case 1. Variable λ_{C_a} occurs in a clustering constraint or same-branching constraint satisfied with equality. All of these constraints are of the form

$$\sum_{C \in \mathcal{V}' : V'' \subseteq C} \lambda_C \geq 1$$

with $V'' \subseteq C_a$, $|V''| = 1$ for the clustering constraints and $|V''| \geq 2$ for the branching constraints. Hence, the theorem holds for $V' = V''$ where V'' corresponds to a constraint satisfied with equality.

Case 2. Otherwise, no clustering constraint or branching constraint containing λ_{C_a} is satisfied with equality. Similarly to Case 3 in the proof of Theorem 5.1, we show by contradiction that this is not possible. Let again $\boldsymbol{\lambda}_B$ be the vector of basic variables, A_B the basic matrix and $\boldsymbol{b}_B$ the corresponding right hand side. λ_{C_a} is fractional and thus a basic variable. Neither the clustering constraints nor the branching constraints containing λ_{C_a} are tight. The variable's upper bound is not satisfied with equality either. Again, the only remaining constraints that might contain λ_{C_a} are the bounds on the number of clusters and we distinguish two subcases:

Case 2.1. A lower or upper bound for the number of clusters is defined and satisfied with equality. As for Theorem 5.1, λ_{C_a} occurs only in a single row of A_B and w.l.o.g. $(\boldsymbol{a}_B)_{C_a} = (0, \ldots, 0, 1)^T$. All entries of A_B and $\boldsymbol{b}_B$ are integers and the last row of A_B consists only of coefficients 0 and 1. Solving equation (5.11) we obtain again $\lambda^*_{C_a} \in \mathbb{Z}$ which contradicts the premise that $\lambda^*_{C_a}$ is fractional.

Case 2.2. No bound for the number of clusters is defined or none of the defined bounds are satisfied with equality. Consequently, it holds $(\boldsymbol{a}_B)_{C_a} = \mathbb{0}$ in contradiction to the premise that $\boldsymbol{\lambda}^*$ is a basic solution.

Consequently, all subcases of Case 2 result in a contradiction and this case cannot occur. In conclusion, the theorem holds. □

Let now $\boldsymbol{\lambda}^*$ be an optimal basic solution with a fractional cluster C_a. According to Theorem 5.2, condition (5.14) holds for some $V'' \subseteq C_a$. Due to $0 < \lambda^*_{C_a} < 1$, a second fractional cluster $C_b \supseteq V''$ exists. Since all clusters are different, there is some node $v \in (C_a \triangle C_b) \subseteq (V \setminus V'')$. As v only occurs in one of the two clusters C_a and C_b, for $V' = V'' \cup \{v\}$ follows

$$0 < \sum_{C \in \mathcal{V}' : V' \subseteq C} \lambda^*_C < 1 \tag{5.15}$$

and additionally $|V'| \geq 2$. Hence, we define the branching constraints based on the node set V'. If multiple sets V'' exists, one of minimal cardinality is selected.

Given a feasible solution of any parent MIP, the sum over any set of cluster variables must be at least zero and integral. Each feasible solution will therefore satisfy either constraint (5.12) or constraint (5.13). Since V is finite, so is $P(V)$. Consequently, after a finite number of branchings no node set $V' \subseteq V$ satisfying the inequalities (5.15) exists. All cluster variables are then integral due to Theorem 5.2 and the branch terminates. Consequently, the branching rule meets all of the criteria stated in Section 2.5.

5.3.2. Branching Constraints in the Pricing Problem

The previous section described how branching constraints are added to the RMP in order to restrict the solution space. These alterations do also affect the PP in different ways. First of all, the PP must prevent the generation of clusters which have been declared infeasible by the branching constraints. Therefore, suitable constraints are added to the PP as well.

Partitioning and Packing

The branching is based on node pairs $\{v_s, v_t\} \subseteq V$. The same-branch ensures that a cluster contains either both nodes together or none of them. For each same-branching constraint (5.8) in the RMP, we add the following equation to the PP:

$$x_s = x_t \tag{5.16}$$

The differ-branch derived from constraints (5.9) prevents clusters from containing both nodes v_s and v_t. In the PP, this is achieved by requiring

$$x_s + x_t \leq 1 \quad . \tag{5.17}$$

Covering

The two branches are based on node sets $V' \subseteq V$ with $|V'| \geq 2$. The same-branch requires the sum over the variables of all clusters that are supersets of V' to be at least 1. However, it does not force any variables to zero. Therefore, only the differ-branch requires amendments to the PP. Here, the same sum is set to zero, excluding each cluster that is a superset of V'. For each differ-branching constraint (5.13) in the RMP, we thus add the inequality

$$\sum_{v \in V'} x_v \leq |V'| - 1 \tag{5.18}$$

to the PP.

5.3.3. Objective Function Adjustments in the Pricing Problem

Recalling equation (5.5), we must additionally adapt the objective function of the PP with respect to the dual variables of the branching constraints in the LPR of the RMP. Let $(\boldsymbol{\pi}_s^*)_i$ be the value of the dual variable corresponding to some branching constraint in the i-th row of the RMP. The term $(\boldsymbol{\pi}_s^*)_i((\boldsymbol{a}_N)_C)_i$ must then be subtracted from the objective function, where C is the cluster computed by the PP.

Partitioning and Packing

Both the same-branching constraints (5.8) and the differ-branching constraints (5.9) sum up all clusters containing a pair of nodes $\{v_s, v_t\} \subseteq V$.

Consider the differ-branching constraints first. Due to inequality (5.17), any feasible cluster C of the PP contains at most one of the nodes v_s and v_t. Thus, it follows $((\boldsymbol{a}_N)_C)_i = 0$ and the objective function of the PP remains unchanged.

This behavior is different for the same-branching constraints. According to equation (5.16), the PP allows for clusters containing either both nodes or neither one of them. If the computed cluster C does contain both nodes, it follows $((\boldsymbol{a}_N)_C)_i = 1$. Otherwise, C does not contain either node and it holds $((\boldsymbol{a}_N)_C)_i = 0$. Consequently, one option is to manipulate the PP's objective function via a new binary variable which takes on value 1 iff

$$((\boldsymbol{a}_N)_C)_i = 1 \Leftrightarrow x_s = 1 \wedge x_t = 1 \quad .$$

However, due to equation (5.16), it further holds $x_s = 1 \wedge x_t = 1 \Leftrightarrow x_s = 1$. The introduction of another variable is therefore unnecessary, since x_s itself already satisfies the required property. Thus, for the same-branch, the objective function is adjusted by subtraction of the term $(\boldsymbol{\pi}_s^*)_i x_s$.

Covering

The branching constraints (5.12) and (5.13) in the RMP sum up all clusters that are a superset of some node set $V' \subseteq V$ with $|V'| \geq 2$.

The differ-branching constraints behave similarly as for the partitioning and packing scenario. Constraint (5.18) prevents any feasible cluster C from containing all the nodes in the set V'. The objective function of the PP remains unchanged due to $((\boldsymbol{a}_N)_C)_i = 0$.

In the same branch, a cluster remains feasible regardless of whether it does contain all the nodes in V' or not. No clusters are declared infeasible by the MP's same-branching constraints and consequently no additional constraints have been added to the PP so far. For each same-branching constraint we add a new binary variable $x_{V'}$ with the following interpretation:

$$x_{V'} \in \{0,1\} = \begin{cases} 1 & \text{if } V' \subseteq C \\ 0 & \text{otherwise} \end{cases}$$

Consequently, the term $(\boldsymbol{\pi}_s^*)_i x_{V'}$ is subtracted in the objective function. In order to ensure that $x_{V'}$ behaves as expected, additional constraints are added to the PP. If $(\boldsymbol{\pi}_s^*)_i$ is positive, $x_{V'}$ will assume the value 0 without further constraints due to the maximization of the objective. Hence, it suffices to add the lower bound

$$\sum_{v \in V'} x_v - |V'| + 1 \leq x_{V'} \tag{5.19}$$

which ensures $x_{V'} = 1$ for $V' \subseteq C$. If to the contrary $(\boldsymbol{\pi}_s^*)_i$ is negative, then $x_{V'}$ will take on the value 1 without additional restrictions. We add the constraint

$$\sum_{v \in V'} x_v \geq |V'| x_{V'} \tag{5.20}$$

to enforce $x_{V'} = 0$ for $V' \nsubseteq C$.

5.4. Branch-and-Price

After having discussed the aspects of pricing and branching, let us briefly summarize the entire branch-and-price approach depicted in Figure 5.1. Given the original problem or MP, we define the RMP on a small subset of the original variables that still render the problem feasible. To solve the MP, we apply a branch-and-bound method complemented with pricing (or column generation) to the RMP.

For each node of the branch-and-bound tree, we check if the LPR of the current RMP is feasible. If not, the branch terminates as usual. Otherwise, the simplex method is employed to compute an optimal solution. Based on the optimal solution's dual values, we next solve the PP to identify variables of positive reduced costs which are not yet part of the RMP because their introduction may improve the objective function value. If such a variable is detected, the LPR is resolved and the pricing loop continues. Otherwise, the optimal solution of the current RMP's LPR corresponds to an optimal solution over all MP variables. If the solution is integral, it is also an optimal solution for the RMP's MIP and the branch terminates. Otherwise, we employ Ryan-Foster branching and add two children to the current node in the branch-and-bound tree. Each branch is based on either a same- or a differ-branching constraint. Since the active branching constraints declare certain clusters infeasible and create new dual variables, they also affect the PP. To prevent the generation of clusters that have become infeasible due to branching, we add further constraints to the PP. Moreover, the objective function of the PP is adapted to account for the new dual variables. Once a node has been processed, another unprocessed node is selected and solved until none are left.

Based on this more detailed understanding, this section examines some additional aspects of branch-and-price.

5.4.1. Farkas Pricing

At the beginning of this chapter we mentioned that the RMP must contain sufficient initial variables to be feasible. As already explained, these variables can be determined, e.g., by some heuristic. However, there are two issues with this approach. First, due to the custom constraints there is no general heuristic than can be applied to arbitrary CVCP instances. Solving different variants of the CVCP would require the implementation of problem-specific heuristics. Moreover, the RMP may result infeasible also during the branch-and-price process due to the insertion of the branching constraints. In this case, it is even more difficult for a heuristic to add suitable variables that turn the RMP feasible again.

Instead, we employ *Farkas pricing* to determine such cluster variables. This technique is based on Farkas' lemma [23]:

Lemma 5.1. *Let $A \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$. Then exactly one of the following two statements holds:*

$$\exists \boldsymbol{\lambda} \in \mathbb{R}^n \; : A\boldsymbol{\lambda} = \boldsymbol{b} \wedge \boldsymbol{\lambda} \geq \mathbb{0} \tag{5.21}$$

$$\exists \boldsymbol{\pi} \in \mathbb{R}^m : A^T\boldsymbol{\pi} \geq \mathbb{0} \wedge \boldsymbol{b}^T\boldsymbol{\pi} < 0 \tag{5.22}$$

The slack form of the RMP is feasible iff statement (5.21) is satisfied for the included variables. Thus, statement (5.22) must hold whenever the RMP is infeasible. The idea is now to add variables in such a way that this statement becomes false. Therefore, we first determine a vector $\boldsymbol{\pi}^*$ that minimizes $\boldsymbol{b}^T\boldsymbol{\pi}$ while satisfying condition (5.22). Note that this corresponds to solving the dual problem with modified coefficients for the objective function. Thus, $\boldsymbol{\pi}^*$ is computed through the simplex method as an optimal solution of a feasible linear program. The vector $\boldsymbol{\pi}^*$ is called a *Farkas multiplier*.

Each Farkas multiplier is a proof for the infeasibility of the RMP. To turn a given proof $\boldsymbol{\pi^*}$ infeasible, it suffices to add a single column $\boldsymbol{a}_C$ to the RMP with $\boldsymbol{a}_C\boldsymbol{\pi}^* < 0$. To determine a feasible column that optimizes a linear objective is precisely the task of the PP. Thus, instead of maximizing the reduced costs, we adapt the objective of the PP to minimize $\boldsymbol{a}_C\boldsymbol{\pi}^*$ and compute the optimal cluster C^*. If $\boldsymbol{a}_{C^*}\boldsymbol{\pi}^* \geq 0$, then statement (5.22) holds over all MP variables and the active branch is infeasible. Otherwise, C^* is added to the RMP to invalidate the current Farkas multiplier. If the RMP is then feasible, branch-and-price continues. Otherwise, we continue to iteratively process the subsequent Farkas multipliers and add further variables to the RMP.

The advantage of this approach is that a solver for the PP can be reused to remove infeasibility from the RMP. Thus, we can apply general purpose MIP solvers to handle any variant of the CVCP.

5.4.2. Upper Bounds

Section 5.2.1 discussed the reduced costs. For a non-basic cluster variable $(\boldsymbol{\lambda}_N)_C$, the corresponding reduced costs $(\bar{c}_N)_C$ state how the objective function value would change if the variable became basic. Since the variable is non-basic, it currently has the value 0. If it turns basic and increases its value by δ, the objective function value increases by $\delta \cdot (\bar{c}_N)_C$.

In the PP, we now determine a variable $(\boldsymbol{\lambda}_N)_{C^*}$ of maximal reduced costs. Consequently, we obtain an upper bound on the reduced costs of all variables. If k_{max} is given, then we also have the upper bound (4.22) on the sum of the cluster variables. Let $\boldsymbol{\lambda}$ be some solution of the current LPR. For the optimal solution $\boldsymbol{\lambda}^*$ over all variables holds:

$$\sum_{C \in \mathcal{V}} c_C \lambda_C \leq \sum_{C \in \mathcal{V}} c_C \lambda_C^* \leq \sum_{C \in \mathcal{V}} c_C \lambda_C + k_{max} (\bar{c}_N)_{C^*} \tag{5.23}$$

Hence, we obtain an upper bound for the optimal objective function value of the current node. The branch-and-bound method can then make use of this bound to speed up the solving process. E.g., it may turn out that the current node can be terminated because there are already primal solutions available that cannot be improved upon.

6. Implementation

This chapter deals with the implementation details of our CVC Framework. The framework is implemented in C++ following the paradigm of object-orientation. Section 6.1 presents the tools and software libraries that were employed for the framework development. The framework architecture is described in Section 6.2. Subsequently, Section 6.3 highlights selected framework features. After presenting the implemented pricers in Section 6.4, the chapter is finally closed with a brief introduction of the concept of initializers in Section 6.5.

6.1. Tools and Libraries

For the development of our framework, we made use of numerous development tools and libraries.

6.1.1. Libraries

For computing optimal clusterings through branch-and-price approaches we chose the open source library SCIP[1] (Solving Constraint Integer Programs). SCIP is a general framework for both mixed integer (linear) programming and mixed integer nonlinear programming (MINLP) that ships with variety of different solvers and even a custom language named ZIMPL[2] to facilitate the modeling of MIPs and MINLPs. It can be run as a standalone command-line application but is also usable as library with interfaces for many common programming languages. The library was chosen for different technical and non-technical reasons. First of all, the Chair of Operations Research at RWTH Aachen University is currently involved in the development of the SCIP library and particularly GCG[3], a SCIP-based generic branch-cut-and-price solver. While GCG is not used within the CVC Framework, the use of SCIP will simplify the understanding, use and extension of the framework for other users at the institute. More importantly, the fact that SCIP is open source allows to manipulate the solving process at its core and thus facilitates the implementation of customized branch-and-price methods. SCIP is implemented in C with additional C++ wrapper classes for user plug-ins. Since we must extend its functionalities for our purposes, we also opted for C++ as the programming language for our framework.

Another major choice was that of an open source C++ graph library with two suitable candidates, the Boost Graph Library[4] on the one hand and the LEMON Graph Library[5]

[1]`http://scip.zib.de/` (last accessed: 08/19/2018)
[2]`http://zimpl.zib.de/` (last accessed: 08/19/2018)
[3]`http://www.or.rwth-aachen.de/gcg/` (last accessed: 08/19/2018)
[4]`http://www.boost.org/doc/libs/1_64_0/libs/graph` (last accessed: 08/19/2018)
[5]`http://lemon.cs.elte.hu/trac/lemon` (last accessed: 08/19/2018)

on the other hand. Boost is by far the more extensive and more powerful option of the two. Nevertheless, we decided to take LEMON because it is a smaller dependency and provides sufficient utilities for our needs.

Neither Boost nor LEMON contain advanced and customizable features for graph visualization so that a further library was required for this purpose. One commonly used and extensive visualization framework is gnuplot[1]. However, gnuplot focuses primarily on the visualization of mathematical functions and data points and provides no graph-specific functionalities. The so-called Open Graph Drawing Framework[2] focuses particularly on graph visualization and provides, e.g., several layout algorithms to automatically compute adequate node positions. We finally selected Graphviz[3] whose features seem to exceed those of the Open Graph Drawing Framework. For instance, Graphviz provides an interactive graph browser and allows to fill nodes with multiple colors, thus making it perfect to illustrate nodes pertaining to multiple subgraphs.

Some smaller libraries were included for specific purposes. We chose spdlog for fast logging[4]. The library yaml-cpp[5] was added to handle YAML files for the configuration of the CVC Framework. Moreover, the framework employs JsonCpp[6] for I/O operations in JSON format.

6.1.2. Integrated Development Environment

To simplify cross platform development on both Windows and Unix systems, Eclipse CDT (C/C++ Development Tooling)[7] was chosen as an IDE.

Doxygen[8] was selected as the standard tool for code documentation which allows the generation of both HTML and LaTeX documentation directly from the source code annotations. Additionally, doxygen is easily integrable into Eclipse via the Eclox plug-in.

Eclipse also contains a test runner which officially supports four different unit testing frameworks, Qt Test[9], Boost Test[10], Google Test[11] and TAP[12] (Test Anything Protocol). TAP was discarded because it was primarily developed for Perl and seems to provide only limited functionalities where as the remaining three are more sophisticated xUnit frameworks. Qt is actually an entire C++ software development framework and its testing component thus targets primarily Qt-based applications. Boost Test and Google Test both provide a wide range of features. We finally opted for the latter, partly because there were some known minor issues with the integration of Boost Test into

[1]`http://www.gnuplot.info/` (last accessed: 08/19/2018)
[2]`http://www.ogdf.net` (last accessed: 08/19/2018)
[3]`http://www.graphviz.org` (last accessed: 08/19/2018)
[4]`https://github.com/gabime/spdlog` (last accessed: 08/19/2018)
[5]`https://github.com/jbeder/yaml-cpp` (last accessed: 08/19/2018)
[6]`https://github.com/open-source-parsers/jsoncpp` (last accessed: 08/19/2018)
[7]`https://eclipse.org/cdt/` (last accessed: 08/19/2018)
[8]`http://www.doxygen.org` (last accessed: 08/19/2018)
[9]`http://doc.qt.io/qt-5/qtest-overview.html` (last accessed: 08/19/2018)
[10]`http://www.boost.org/doc/libs/1_46_1/libs/test` (last accessed: 08/19/2018)
[11]`https://github.com/google/googletest` (last accessed: 08/19/2018)
[12]`http://testanything.org/` (last accessed: 08/19/2018)

Eclipse.

6.2. Framework Architecture

With the background knowledge of the employed libraries, let us examine the architecture of the CVC Framework.

6.2.1. Main Packages

The CVC Framework is divided into seven main packages which are illustrated by figure 6.1.

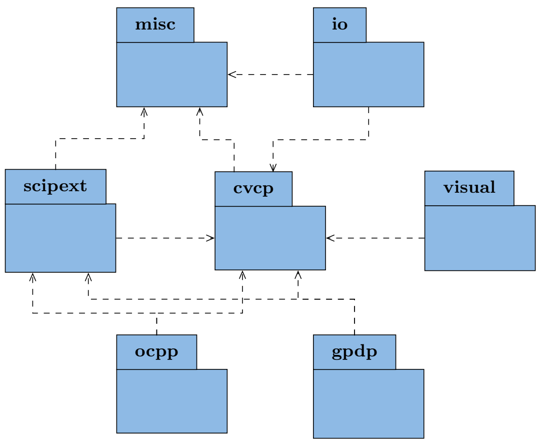


Figure 6.1.: *UML package diagram of the top-level packages of the CVC Framework. Only the principal dependencies are shown.*

The package `cvcp` contains all central data structures for representing a CVCP. The package `misc` unites a variety of low-level functionalities like hash functions and associative containers for LEMON objects, custom exceptions, general constants and simple utility functions. The package `io` primarily provides input and output functionalities for graphs, MIPs, CVCPs and CVCP solutions, but also access to loggers and the framework configuration. SCIP extensions like a generalized version of the Ryan-Foster branching rule as described in Section 5.3 and different solvers for the pricing problem are contained in the package `scipext`. The package `visual` allows the visualization of graphs and node clusterings. Finally, the packages `ocpp` and `gpdp` are extensions to CVC Framework for handling the Odd Cycle Packing Problem and the German Political Districting Problem, two special CVCP variants that are described in Chapters 7 and 8.

6.2.2. Core Classes

Figure 6.2 displays the core classes involved in the solving process of a CVCP. Each of these classes belongs to either the `cvcp` or the `scipext` package. Note that class information may be incomplete and that certain types, attributes and functions were

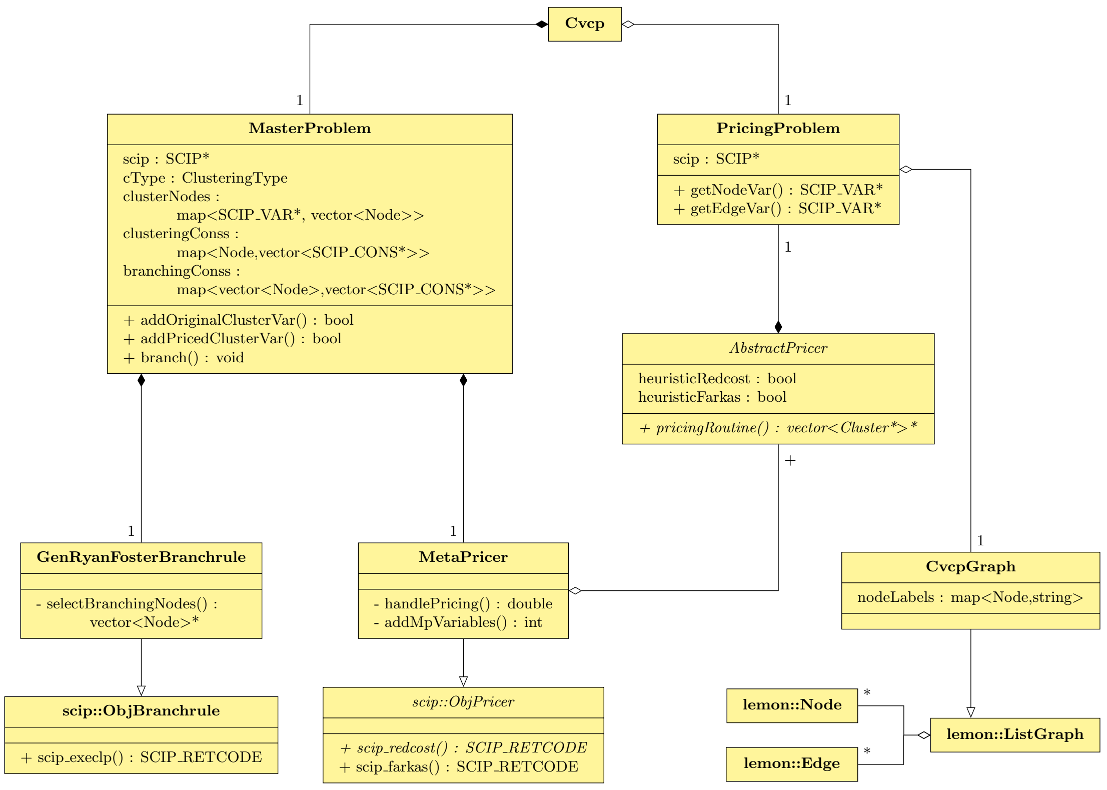

Cvcp
1
MasterProblem
scip : SCIP*
cType : ClusteringType
clusterNodes :
map<SCIP_VAR*, vector<Node>>
clusteringConss :
map<Node,vector<SCIP_CONS*>>
branchingConss :
map<vector<Node>,vector<SCIP_CONS*>>
+ addOriginalClusterVar() : bool
+ addPricedClusterVar() : bool
+ branch() : void
1
PricingProblem
scip : SCIP*
+ getNodeVar() : SCIP_VAR*
+ getEdgeVar() : SCIP_VAR*
1
AbstractPricer
heuristicRedcost : bool
heuristicFarkas : bool
+ pricingRoutine() : vector<Cluster*>*
+
1
GenRyanFosterBranchrule
- selectBranchingNodes() :
vector<Node>*
1
MetaPricer
- handlePricing() : double
- addMpVariables() : int
1
CvcpGraph
nodeLabels : map<Node,string>
scip::ObjBranchrule
+ scip_execlp() : SCIP_RETCODE
scip::ObjPricer
+ scip_redcost() : SCIP_RETCODE
+ scip_farkas() : SCIP_RETCODE
lemon::Node
*
lemon::Edge
*
lemon::ListGraph

Figure 6.2.: *UML class diagram of the core classes of the CVC Framework. Classes that form part of the SCIP or LEMON library are prefixed by the `scip::` or `lemon::` corresponding namespace identifier. Attributes are considered private.*

renamed for compactness and illustration purposes. E.g., the names of smart pointers[1,2] were replaced by the raw pointer symbol (asterisk). Moreover, attributes are considered to be private and to have a corresponding getter-method.

Any problem instance is represented by an object of class `Cvcp` which is composed of two attributes, a `MasterProblem` and a `PricingProblem`.

The `PricingProblem` contains a `SCIP` instance which models the PP as a MIP according to Section 5.2.2. Only the connectivity constraints are left out. Instead, the connectivity data is stored in a `CvcpGraph` object. This class inherits from the LEMON class `ListGraph`, which represents an undirected graph comprising LEMON `Node` and `Edge` attributes. For each node, a corresponding label is stored in the `CvcpGraph`. The `PricingProblem` makes use of this label to map nodes and edges to their corresponding variables. In SCIP, each variable is identified by a name. Thus, for a node with label *node1*, the corresponding node variable in the `SCIP` instance of the PP is named *x$node1*. Given an edge to a second node with label *node2*, the corresponding edge variable is name *y$node1$node2*. The `PricingProblem` facilitates the access to these node and edge variables through convenience getter-methods.

The aggregated extended formulation of the CVCP, i.e., the MP, is stored in the `SCIP` instance of the `MasterProblem`. Additionally, this class contains an enum

[1]`https://en.cppreference.com/w/cpp/memory/unique_ptr` (last accessed: 08/19/2018)
[2]`https://en.cppreference.com/w/cpp/memory/shared_ptr` (last accessed: 08/19/2018)

ClusteringType which indicates if a packing, partitioning or covering must be computed. A mapping assigns each node the corresponding clustering constraint.

The SCIP is solved by calling the SCIP method SCIPsolve() on the SCIP object of the MasterProblem. The solving method executes the branch-and-price approach and invokes the branching rule and pricers. Although SCIP is written in C, it provides C++ wrapper classes to facilitate the implementation of plug-ins. Such plug-ins can then be registered with a SCIP instance and will be applied thereafter in the solving process via callback methods.

For the CVC Framework, we implemented Ryan-Foster branching through the GenRyanFosterBranchrule as a subclass of SCIP's ObjBranchrule. To perform branching, the callback method scip_execlp() is invoked. The custom branching rule then determines which nodes to branch on via the method selectBranchingNodes(). Through the method branch(), the nodes are passed to the MasterProblem where the branching constraints are created and added to the branchingConss mapping. Finally, the resulting subproblems are added to the branch-and-bound tree.

Similarly, the MetaPricer inherits from the abstract SCIP wrapper class ObjPricer and is called upon in the pricing loop to solve the PP. The callback methods are scip_redcost() and scip_farkas() for reduced cost pricing and Farkas pricing, respectively. However, the PP is not solved by the MetaPricer itself, but by instances of the abstract class AbstractPricer. The MetaPricer holds at least one instance of this class. To solve the PP, its method handlePricing() is executed. Then, iteratively, it calls for each AbstractPricer the abstract method pricingRoutine() to determine suitable clusters for which a variable should be added to the MP. As soon as some AbstractPricer returned a cluster, the MetaPricer discards the remaining ones for this pricing round. The method addMpVariables() is executed and cluster variables are added to the MasterProblem by calling addPricedClusterVar(). The nodes corresponding to each cluster variable are stored in the mapping clusterNodes.

The utilization of AbstractPricer instances allows to apply different solvers to the PP. Each pricer runs on a separate copy of the PricingProblem and has two boolean attributes heuristicRedcost and heuristicFarkas. The attributes indicate whether the pricer solves the corresponding type of PP heuristically or exactly. If a cluster of positive reduced cost or Farkas value exists, then an exact pricer must always return such a cluster. A heuristic pricer is not obligated to do so and may return an empty vector instead. Consequently, even if an exact pricer returns no cluster, the AbstractPricer discards the remaining pricers because there is no variable to add to the MP. At least one exact pricer should be applied for each type of pricing. Note that it is not necessary to add the optimal cluster as defined by equation (5.5) to the MP. Any cluster of positive value works. Moreover, multiple such clusters may be added at once.

A concrete implementation of AbstractPricer may solve the PP in any arbitrary way. One option is to use SCIP on the corresponding MIP formulation that is already largely given by SCIP instance in the copy of the PricingProblem. Since the connectivity constraints are not present in the original instance, it is then the pricer's task to add them. This can be accomplished, e.g., by following one of the approaches from Section 3.3. Different pricer implementations for general CVCP instances as well as certain CVCP variants are introduced in Sections 6.4, 7.2 and 8.2.

6.2.3. Plug-In Architecture

Although the CVC Framework is capable of solving arbitrary CVCP instances, it may be desirable to extend its functionalities when dealing with variants of the problem. A plug-in architecture facilitates the extension of certain framework components. This concept is illustrated by Figure 6.3 on the example of pricer implementations. For illustration purposes, we assume that some application extends the CVC Framework with additional classes.

As already described in Section 6.2.2, any PP solver must inherit from the class `AbstractPricer`. To ease the management of specific pricer implementations, the Gang of Four's *factory method pattern* was implemented [27].

`BaseFactory` is a factory class template used for creating not only pricers but also other components. There are three template parameters. The parameter `T` states a (typically abstract) type of objects to be created. The parameter `K` defines the type of key by which a concrete implementation of `T` is identified. Finally, the parameter `P ...` corresponds to an arbitrary amount of types of objects that are used for the construction of an instance of type `T`. In a `registry` map, the template maps each key to the function pointer of a builder function. The builder function is passed the arguments `P` in order to create a suitable instance of the `T` implementation specified by the key and return its pointer. To obtain an instance of `T`, the method `getObject()` is called with

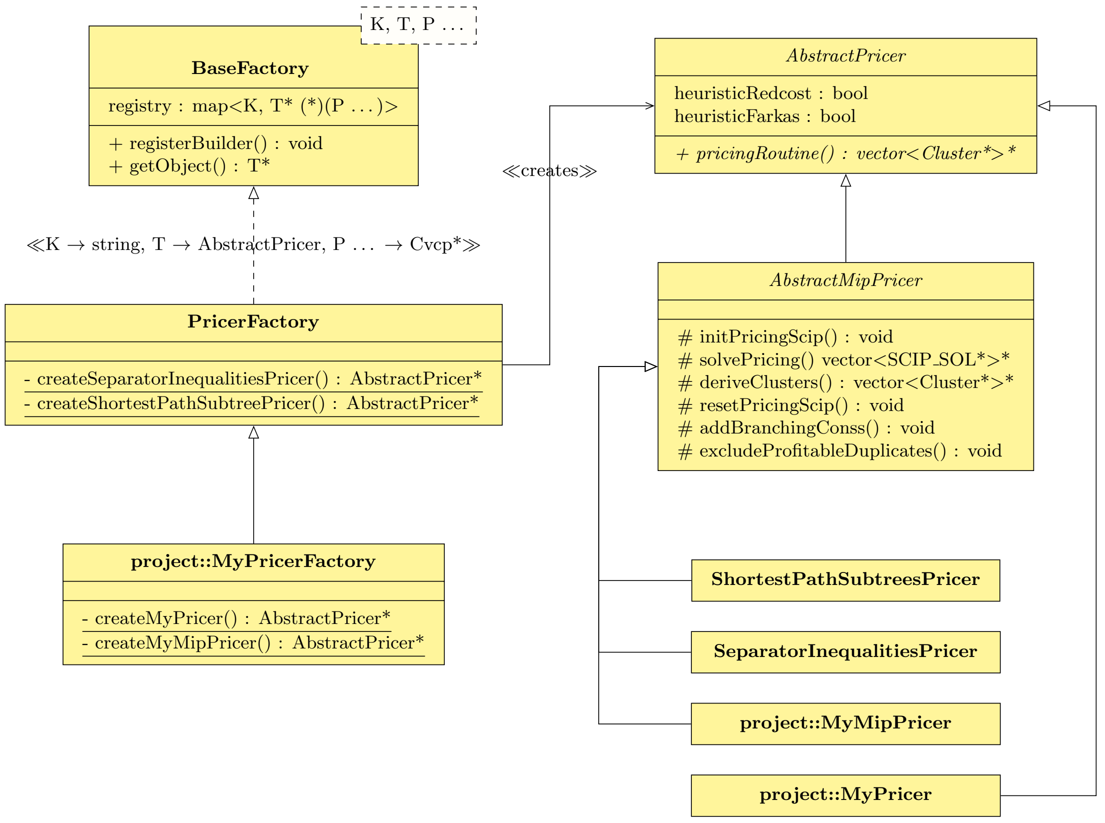

K, T, P ...
BaseFactory
registry : map<K, T* (*)(P ...)>
+ registerBuilder() : void
+ getObject() : T*
≪K → string, T → AbstractPricer, P ... → Cvcp*≫
PricerFactory
- createSeparatorInequalitiesPricer() : AbstractPricer*
- createShortestPathSubtreePricer() : AbstractPricer*
project::MyPricerFactory
- createMyPricer() : AbstractPricer*
- createMyMipPricer() : AbstractPricer*
≪creates≫
AbstractPricer
heuristicRedcost : bool
heuristicFarkas : bool
+ pricingRoutine() : vector<Cluster*>*
AbstractMipPricer
# initPricingScip() : void
# solvePricing() vector<SCIP_SOL*>*
# deriveClusters() : vector<Cluster*>*
# resetPricingScip() : void
# addBranchingConss() : void
# excludeProfitableDuplicates() : void
ShortestPathSubtreesPricer
SeparatorInequalitiesPricer
project::MyMipPricer
project::MyPricer

Figure 6.3.: *UML class diagram of the plug-in architecture of the CVC Framework. Extensions of the framework are marked with the* `project::` *namespace identifier.*

parameters of type `P ...` and `K`. This method then uses the `registry` to identify the correct builder function and invoke it. Builder functions can be added to the factory dynamically by passing their function pointer and the intended key to the method `registerBuilder()`. This setup a common interface with flexible means of creating suitable object instances for different application scenarios.

The right hand side of Figure 6.3 depicts different pricer implementations. `AbstractMipPricer` is an abstract subclass of `AbstractPricer` that eases the implementation of MIP-based pricers. It provides different functionalities like initialization and resetting methods for the `SCIP` instance of the PP, insertion of branching constraints, a solving method and a method to derive the clusters corrsponding to a SCIP solution. An inheriting class must essentially just insert the connectivity constraints into the `SCIP` instance before calling the solving method. Two corresponding subclasses, the `ShortestPathSubstreesPricer` and the `SeparatorInequalitiesPricer` are shipped with the CVC Framework. They are based on the approaches from Section 3.3 and described more in detail later in Section 6.4. The application adds two further implementations `MyMipPricer` and `MyPricer`. The first inherits from `AbstractMipPricer`, the second from `AbstractPricer` directly.

The creation of a pricer is now delegated to a corresponding factory. Due to the factory, the framework can create pricers without actually being aware of the existing implementations. The class `PricerFactory` is a specialization of the `BaseFactory` template for the creation of `AbstractPricer` instances. It employs `string` IDs and `Cvcp` pointers as builder parameters. Two builder functions are already preregistered. The application has now different options for including its own pricers into a factory. First, a new class `MyPricerFactory` may be added that inherits from `PricerFactory` and provides additional builder functions for the custom pricers. Alternatively, the `registerBuilder()` method allows to add pricers to a factory dynamically at runtime.

The factory method pattern is not only applied for the creation of `AbstractPricer` instances, but also for other classes like graph and MIP readers. This allows, e.g., to add capabilities for handling new file formats without the necessity of changing any existing code.

6.3. Features

In addition to the architectural aspects, let us present some features of the CVC Framework.

The previous section already described the framework's plug-in architecture and mentioned that the plug-in concept extends, e.g, to file readers. Table 6.1 lists which input and output formats are currently supported. The LEMON Graph Format (.lgf)

Table 6.1.: *Supported I/O formats.*

Data Type	Input Formats	Output Formats
graphs	.lgf	.lgf
MIPs	.lp, .zpl	
CVCP solutions		.json

consists of a node list and an edge list with the option to pass additional node, edge, and graph attributes. The CPLEX LP file format (.lp) is a simple notation for linear programs with separate sections for, e.g., the objective, constraints and bounds. ZIMPL (.zpl) is a file format for linear and nonlinear mixed integer programs with more language constructs for short and human-readable formulations. If required, further formats can easily be added.

Aside from data input and output, the CVC Framework is also capable of visualizing graphs and CVCP solutions, i.e., clusterings. Figure 6.4 depicts a clustering visualization generated by the framework.

The clustering comprises 4 disjoint clusters, each one indicated by a different color. For coverings, where nodes' may belong to multiple clusters, each node is depicted as a pie chart that represents all cluster affiliations. Visualizations of graphs and solutions are stored as SVGs with tooltips for additional information such as objective function coefficients.

To support users and developers in the analysis of the solving process, the framework logs data on all algorithm executions. Python modules are provided to facilitate the access to this data and to create a variety of corresponding plots. These plots are shown in appendix A. Figure A.1 displays the change of the objective bounds over time. The plot of Figure A.2 states for each pricing round the duration and how many variable were added to the RMP. The subsequent Figure A.3 breaks this information down even further to the level of each single pricer call. Figure A.4 provides a briefer overview on which pricers succeeded in inserting variables into the RMP. The last plot in Figure A.5 indicates at which stages of the solving process the variables of the optimal root LP and root IP solution were integrated into the RMP.

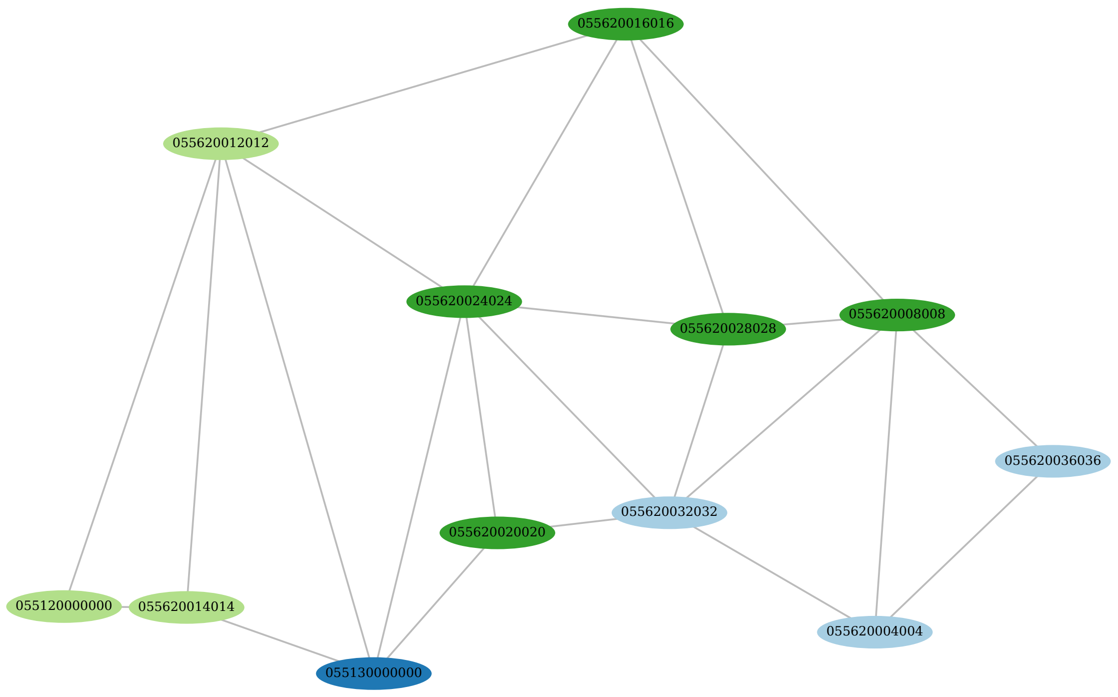


Figure 6.4.: *Clustering visualization generated by the CVC Framework.*

Finally, the framework eases the implementation of additional pricers through different options for handling cluster regeneration. In reduced costs pricing, an optimal solution of the PP provides the reduced costs $(\bar{c}_N)_{C^*}$ of the cluster variable λ_{C^*} to add to the RMP. Since the values $\boldsymbol{\pi}_s^*$ of the RMP's dual variables are known, the correct objective function coefficient $(c_N)_{C^*}$ for the MP is then derived through the equations in (5.4). However, it is sufficient to add any variable of positive reduced costs to the RMP. In practice, the PP is thus often not solved to optimality. Instead, the pricer stops, e.g., as soon as a variable of positive reduced cost was determined. For some CVCPs, the encoding of a cluster in the PP is not unique, i.e., due to the custom variables multiple feasible solutions may correspond to the same cluster. This is the case, e.g., for the GPDP presented later in Chapter 8. In contrast to equation (4.19), the custom variables $\boldsymbol{y}$ might then not maximize the objective function value for the cluster defined by $\boldsymbol{x}$. Deriving the MP coefficient as described above may then result in a value lower than the true coefficient. One consequence a later pricer execution might recompute the same cluster again with an improved objective function value and we obtain duplicate variables in the MP. The CVC Framework implements four different strategies for handling this issue.

For many problems, the encoding of the clusters is unique. In this case, the recomputation of a cluster by a pricer indicates some error in the implementation. Thus, the *abort* strategy causes a runtime error for any attempt to add a duplicate cluster variable to the RMP. For partitioning and packing problems, duplicates cluster variables in the RMP are not an issue because the clustering constraints prevent all feasible clusterings from containing multiple equivalent clusters. Moreover, if any cluster with duplicate variables occurs in an optimal solution, it will always be represented by the variable with the highest objective function coefficient. Thus, the strategy *allow duplicates* accepts the insertion of duplicate variables without taking any special measures. Similarly, the *replace duplicates* strategy removes the old variable of lower coefficient from the RMP and adds a new one instead. However, as explained in Section 5.2.2, cluster regeneration may be explicitly prevented for covering problems through the exclusion constraints (5.6). In this case, it must be ensured that all variables are directly added with the correct coefficient. Therefore, the last strategy *cluster evaluator* allows to pass an object to the solver that is capable of computing the true coefficient of any given cluster. The strategy may be used for non-covering problems as well to speed up the solution process.

6.4. Pricers

Section 6.2.3 already stated that the CVC Framework includes two different pricer implementations. Both inherit from the class `AbstractMipPricer` and thus primarily have the task of creating suitable connectivity constraints.

6.4.1. Shortest Path Subtrees Pricer

The *Shortest Path Subtrees Pricer (SPSP)* follows the approach from Section 3.3.3 to derive connectivity constraints from shortest path subtrees. As already explained there, the

PP is then not just a single MIP. Instead, for each node $v \in V$ a different subproblem must be solved where v is selected as the center node v_c. In order to reduce execution times, we generally do not solve a subproblem for all the nodes $v \in V$. Instead, we stop pricing after the first subproblem that determined a cluster of positive reduced costs. Although it may not be optimal, the corresponding variable is then added to the MP and the pricing loop continues.

In order to ensure that the center node is part of the computed cluster, the corresponding node variable is fixed to 1. The definition of the connectivity constraints then requires the computation of the shortest path predecessors $N^<(v, v_c)$ for each pair of nodes $(v, v_c) \in V \times V$. For a given center v_c, the predecessors of all other nodes are computed in time $\mathcal{O}(n^2)$ with a slightly modified Dijkstra algorithm. The use of a suitable priority queue like a Fibonacci heap or Brodal queue would allow to reduce the time complexity to $\mathcal{O}(n \log(n) + m)$ [11, 26]. However, the C++ standard library does not provide such a container and the quadratic runtime is not critical for the branch-and-price approach as a whole.

Note that there are certain clusters which are connected but still infeasible due to the constraints (3.13) of the SPSP. One example is illustrated by Figure 3.1. Consequently, the SPSP is a heuristic pricer because it cannot compute arbitrary connected clusters.

6.4.2. Separator Inequalities Pricer

The *Separator Inequalities Pricer (SIP)* ensures connectivity through the separator inequalities (3.11) introduced in Section 3.3.2. This requires one constraint for each minimal v_a, v_b-separator $\tilde{V}$ of two non-adjacent nodes $v_a, v_b \in V$. Adding all these inequalities

Algorithm 6.1 Node Separators

```
function COMPUTESEPARATORS($C$)
    $separators = \emptyset$
    $clusterComps \leftarrow$ COMPUTECONNECTEDCOMPONENTS($G[C]$)
    for $c_a \in clusterComps$ do
        $v_a \leftarrow c_a[0]$
        $complementComps \leftarrow$ COMPUTECONNECTEDCOMPONENTS($G[V \setminus c_a]$)
        for $complementComp \in complementComps$ do
            for $v \in complementComp$ do
                $reachable[v] \leftarrow complementComp$
        $separator = \emptyset$
        for $c_b \in clusterComps \setminus c_a$ do
            $v_b \leftarrow c_b[0]$
            for $v \in c_a$ do
                for $\{u, v\} \in E$ do
                    if $u \notin c_a \wedge reachable[u] = reachable[v_b]$ then
                        $seperator \leftarrow separator \cup \{u\}$
            $separators \leftarrow separators \cup \{(v_a, v_b, separator)\}$
    return $separators$
```

to the PP might result in a number of constraints that is exponential in n. This has a significant impact on the efficiency of the pricer. To circumvent this issue, we opt for a branch-and-cut approach, i.e., lazy constraint insertion. The PP is solved at first without any connectivity constraints. As suggested by Fischetti et al., we check whether the computed cluster is connected whenever an integral PP solution is determined [25]. Only if this is not the case, we turn the detected solution infeasible by adding separator inequalities to the pricing problem. For a given unconnected cluster C, we employ Algorithm 6.1 to compute suitable node pairs $v_a, v_b \in V$ and corresponding separators $\tilde{V}$. One separator inequality is added for each pair of connected components $c_a, c_b \subseteq C$ of the solution cluster.

6.5. Initializers

Another type of CVC Framework plug-in is the *initializer*. Initializers have the task to add variables to the RMP before the branch-and-price process starts. The idea is to efficiently determine promising clusters and speed up the execution by reducing the number of pricer calls later in the solving process. The framework does not provide any general purpose initializers because a useful implementation is highly dependent on the given problem. Some initializers for different CVCP variants are introduced in the following two chapters.

7. Odd Cycle Packing Problem

For a graph G, the OCPP consist of finding a node packing of maximum cardinality where each cluster forms a cycle of odd length [39]. The cardinality of an optimal packing is also called *odd cycle packing number (OCPN)*. One application of the OCPP is the NP-hard *Stable Set Problem (SSP)*, also known as the *Independent Set Problem*. The SSP consists of finding a subset of nodes $V' \subseteq V$ of maximum cardinality in a given graph G which satisfies the condition that no pair of nodes $v_i, v_j \in V'$ is adjacent. For a graph with $OCPN \in \mathcal{O}(n/\log n)$, the SSP can be approximated with a constant approximation factor in polynomial time [9].

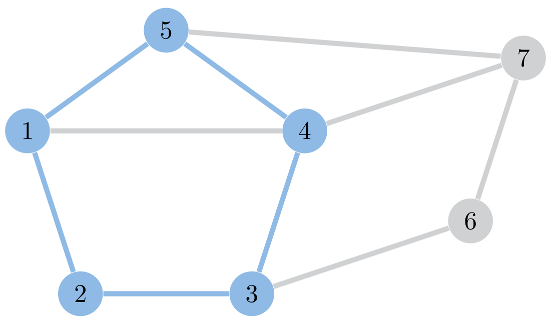


Figure 7.1.: *Non-induced odd cycle subgraph (marked blue).*

Note that the clusters in the CVCP correspond to node-induced subgraphs, but a cycle is not necessarily a node-induced subgraph. Consider, e.g., Figure 7.1 as an example. The blue nodes $V' = [5]$ and the blue edges form a cycle subgraph C of odd length. However, the node-induced subgraph $G[V']$ is not a cycle due to the extra edge $\{1,4\}$. For a cycle subgraph C of some graph G, such an edge that is not part of $E(C)$ but belongs to the node induced subgraph $G[V(C)]$ is called a *chord* of C. A cycle without chords is said to be *chordless*.

To model the OCPP as a CVCP, we could declare all clusters feasible that contain a cycle of odd length, even if the node-induced subgraph itself is not a cycle. This would compute the OCPN correctly, but would require further post-processing to determine the odd cycles. Instead, we consider a cycle of odd length to be feasible iff it is node-induced, i.e., chordless. Cycles with chords like the one from Figure 7.1 are considered infeasible.

However, any odd cycle can be transformed into a chordless one by removing some of its nodes. Assume that C is an odd cycle with nodes V' and edges E' given by:

$$
\begin{aligned}
V' &= \{v_1, \ldots, v_{n'}\} \\
E' &= \{\{v_1, v_2\}, \ldots, \{v_{n'-1}, v_{n'}\}, \{v_{n'}, v_1\}\}
\end{aligned}
$$

Let there be a chord $\{v_a, v_b\} \in E \setminus E', \{v_a, v_b\} \subseteq V'$. The nodes v_a and v_b are not adjacent in C because otherwise the edge $\{v_a, v_b\}$ would be part of the cycle and not a chord. W.l.o.g., let $a = 1$ and therefore $b \in [3, n' - 1]$. Thus, the node set $\{v_a, v_b\}$ is a separator for C. More precisely, it separates the cycle into exactly two different connected components with nodes $V_1' = \{v_2, \ldots, v_{b-1}\}$ and $V_2' = \{v_{b+1}, \ldots, v_{n'-1}\}$, respectively. Both components consist of at least one node and one of them must be of odd cardinality since exactly two nodes were removed from the odd cycle C. We may assume w.l.o.g. that V_1' has odd cardinality. Using the edges from C and the chord $\{v_a, v_b\}$, the nodes $V'' = V_1' \cup \{v_a, v_b\}$ form an odd cycle. V'' is a proper subset of V'. Consequently, we can transform any odd cycle C with a chord into a shorter odd cycle whose nodes are a subset of $V(C)$. Repeating the transformation as long as a chord is present, one eventually obtains a chordless odd cycle.

To illustrate the transformation, let us have a second look at the highlighted odd cycle from Figure 7.1. Removing the nodes 1 and 4 results in the two connected components $\{5\}$ and $\{2, 3\}$ of which the first one has odd cardinality. Joining this component with the nodes from the chord results in a shorter odd cycle with the nodes $\{1, 4, 5\}$.

Let now $\mathcal{C}^* = \{C_1, \ldots, C_{n_c^*}\}$ be an odd cycle packing of maximum cardinality that potentially includes cycles with chords. We transform each cluster (or cycle) $C_k \in (\mathcal{C}^*$ into a chordless cycle C_k' with $V(C_k) \subseteq V(C_k)$. Since the clusters in $\mathcal{C}^*$ are pairwise node-disjoint, so are the corresponding subsets C_k'. Then $\mathcal{C}' = \{C_1', \ldots, C_{n_c^*}'\}$ is also a maximal odd cycle packing. Consequently, there always exists an optimal solution that only consist of cycles which are chordless and therefore node-induced. Restricting the feasible clusters to node-induced odd cycles may therefore reduce the space of feasible solutions but will not alter the feasibility or optimal objective value of any instance.

Table 7.1.: *CVCP configuration for the OCPP.*

Input	Value
o_type	*max*
c_type	*packing*
G	input graph
$\boldsymbol{c}_x$	$\mathbb{0}$
k_{min}	0
k_{max}	$\lfloor \frac{n}{3} \rfloor$

The configuration of the CVCP for OCPP instances is stated in Table 7.1. The objective function that we define later is to be maximized and is not directly affected by the values of the node variables, i.e., it holds $\boldsymbol{c}_x = \mathbb{0}$. Since we assume graphs to be simple, there cannot be any loops and any odd cycle is of length at least 3. Moreover, each node appears in at most one cycle. Consequently, we may define k_{max} as $\lfloor \frac{n}{3} \rfloor$. The custom variables with their costs and integer constraints together with the custom constraints are defined in the following.

7.1. Pricing Problem

In addition to the node variables x_i, we employ edge variables y_{ij} as defined in equation (3.4). To ensure that the variables y_{ij} assume the correct values, we also add the constraints (3.5) through (3.7). The computation of odd cycles requires further custom constraints.

First, we define the following cycle constraints:

$$\sum_{\{v_i,v_j\}\in E} y_{ij} = 2x_i \quad \forall v_i \in V \tag{7.1}$$

Let $G' = (V', E')$ denote the computed subgraph with $V' \neq \emptyset$. It holds $n' = |V'| \geq 3$ because any simple graph with fewer nodes contains at most one edge and thus cannot not satisfy the constraints (7.1). The equations require each node of the subgraph to be incident to exactly two edges. Since there are no loops and each edge is counted once for each end node, the computed subgraph satisfies $|E'| = n'$. A connected simple graph with n edges and $n \geq 3$ contains exactly one cycle. This also applies to G' and moreover that cycle must be G' itself. Otherwise, there exists a proper subgraph G'' of G' that forms a cycle. As G'' is a cycle, each of its nodes is incident to two edges. Moreover, there must be a node $u \in V(G'')$ that is adjacent to a note $v \in V(G') \setminus V(G'')$ because G' is connected. Consequently, u is incident to at least 3 edges in G'. Due to constraint (7.1) this is not possible and so G' itself is indeed a cycle.

For the argument to hold, we must ensure that $V' \neq \emptyset$. This is already implied if we only allow odd cycles. To ensure that the computed cluster has odd cardinality, we add an auxiliary variable $z_{card} \in \mathbb{N}$ and the following constraint:

$$\sum_{v_i\in V} x_i = 2z_{card} + 1$$

Finally, we define the objective function. For the MP, the objective is to maximize the size of the packing and determine the OCPN. Therefore, each cluster in the MP must have coefficient 1. Hence, we add a constant $z_{const} = 1$ to the PP and define a constant objective function:

$$\max z_{const}$$

7.2. Framework Plug-Ins

In order to speed up solving for OCPP instances, we extended the CVC Framework with two problem-specific plug-ins. Both are based on Algorithm 7.1 which computes all cycles of length 3 for a given graph $G = (V, E)$.

Section 6.5 already explained the concept of initializers. For the CVCP, we implemented a *Three-Cycle Initializer (TCI)* that adds all cycles of length 3 to the RMP. Additionally, we added a *Three-Cycle Pricer (TCP)* that is heuristic. Note that even odd cycles may become locally infeasible due to the branching constraints. Upon being called, the TCP thus determines all feasible three-cycles of positive reduced costs. Then a fixed percentage of cycles with highest reduced costs is selected and the corresponding variables are inserted into the RMP.

Algorithm 7.1 Three-Cycles

```
function COMPUTETHREECYCLES($G$)
    $cycles \leftarrow \emptyset$
    for $v_i \in V$ do
        for $\{v_i, v_j\} \in E : j > i$ do
            for $\{v_i, v_k\} \in E : k > j$ do
                if $\{v_j, v_k\} \in E$ then
                    $cycles \leftarrow cycles \cup \{(v_i, v_j, v_k)\}$
    return $cycles$
```

8. German Political Districting Problem

As already stated in Section 1.2, the CVCP was originally motivated by the GPDP and is a generalization thereof. Modeling and solving techniques for political districting problems have been studied for decades [34, 35, 47, 60]. A lot of this research focuses on the issue of Gerrymandering, as presented in Section 1.1. However, we focus on the German federal elections where Gerrymandering is a less critical due to the voting system. Instead, other aspects must be taken into account.

The aim of the GPDP is the computation of optimal constituencies for the German federal elections. The Federal Elections Act (German: Bundeswahlgesetz, BWG) states multiple guidelines for the definition of the constituencies [30, §3].

One requirement is that the constituencies must obey the federal states' boundaries, i.e., no constituency may be part of multiple states. This simplifies the districting process because the number of each state's constituencies depends on its population and is known in advance. Consequently, districting may be performed for each federal state individually, thus reducing the size of the overall problem. Additionally, the German population of a constituency may not deviate by more than 25% from the constituencies' average German population. Moreover, constituencies must be connected, i.e., for any two arbitrary points A and B of a constituency it must be possible to reach B from A without leaving the constituency.

Besides these hard constraints, the BWG states some more objectives for the attributes of the constituencies. If possible, in addition to the population bound above, the constituency population should not differ from the average by more than a tolerance limit of 15%. Furthermore, constituency boundaries should match the administrative boundaries of municipalities, districts and so-called district-free cities, an administrative sub-division in Germany. Although not explicitly stated by law, further objectives are being considered in practice. One goal is to balance the constituency population in order to ensure that all voters have equal influence on the election outcome. Additionally, it is inconvenient for both voters and the organization of the elections when a constituency is altered from one election to the next. Consequently, the continuity of constituencies is taken into account as well, i.e., it is attempted to leave constituencies unchanged.

Recent research has already examined the task of political districting for German elections in particular [10, 31, 32]. Taking into account the previous requirements, Goderbauer and Lübbecke formalized the GPDP in terms of a MIP [33]. Note that in accordance with the previous considerations, the districting is performed for each single state separately. For any given state, it is assumed that a set $V = \{v_1, \dots, v_n\}$ of indivisible base population units as well as a fixed target amount n_c^* of constituencies is given. Using V as a node set, we define the base unit graph $G = (V, E)$ with edges

$$\{v_i, v_j\} \in E \Leftrightarrow \text{base units } v_i \text{ and } v_j \text{ share a common border} \quad . \tag{8.1}$$

A connected constituency then corresponds to a connected cluster $C \subseteq V$.

This chapter describes the GPDP as a specialization of the CVCP. Just like for the OCPP, we use GPDP instances to evaluate the implemented framework. As in the previous chapter, we therefore first provide the original formulation of the GPDP and proceed with the presentation of GPDP-specific initialization heuristics and pricers.

Table 8.1.: *CVCP configuration for the GPDP.*

Input	Value
o_type	*max*
c_type	*partitioning*
G	base unit graph
$\boldsymbol{c}_x$	$\mathbb{0}$
k_{min}	n_c^*
k_{max}	n_c^*

Table 8.1 shows how to configure the CVCP for GPDP instances. The GPDP is a partitioning problem since each base unit should occur in exactly one constituency. As for the OCPP, the objective function is to be maximized and is not directly affected by the node variables. Since the number of constituencies is explicitly given, it holds $k_{min} = k_{max} = n_c^*$. The GPDP's custom variables with corresponding costs and integer constraints as well as the further custom constraints are defined in the following section.

8.1. Pricing Problem

Like for the OCPP in Section 7.1, we define edge variables y_{ij} with corresponding constraints in addition to the node variables x_i. The remaining custom variables and constraints and the objective function are introduced in the subsequent subsections.

8.1.1. Objective Function

Defining the objective function requires quantifications of all optimization goals. Therefore, metrics with value range 0 to 1 are employed to measure compliance with the objectives. Corresponding metric variables are defined with respect to each of the four goals and for each constituency. Essentially, the better an objective is fulfilled, the higher is the corresponding metric value. The objective function is then just a weighted sum of the metric variables which is to be maximized.

Population Tolerance

Whether a constituency meets the tolerance limit for the population is indicated through the binary variable $z^{tol} \in \{0, 1\}$ that is subject to the following inequalities:

$$-0.15 - 0.1 \cdot (1 - z^{tol}) \leq \frac{\sum_i x_i p_i}{\overline{p}^{const}} - 1 \leq 0.15 + 0.1 \cdot (1 - z^{tol}) \tag{8.2}$$

Here, parameter p_i is the German population of base unit v_i and $\overline{p}^{const}$ the average German population of a constituency. Note that this average constituency population is not computed separately for each state, but for Germany as a whole. Due to the maximization of the objective function, z^{tol} equals zero iff the German population of the constituency deviates from the average by more than 15%, thus exceeding the tolerance limit. Simultaneously, constraint (8.2) guarantees that the absolute deviation limit of 25% is obeyed.

Adherence to Administrative Boundaries

Variable $z^{bound} \in [0, 1]$ determines in how far the computed constituency obeys administrative boundaries. First, we define an auxiliary variable $z_p \in [0, 1]$ that states which percentage of the constituency's boundary is also a district boundary. The following parameters are used for its definition:

b_i	$=$	length of the boundary of base unit v_i
b_i^{dist}	$=$	length of the boundary of v_i corresponding to the boundary of a district or district-free city
b_{ij}	$=$	length of the shared boundary between base units v_i and v_j
b_{ij}^{dist}	$=$	length of the shared boundary between base units v_i and v_j corresponding to the boundary of a district or district-free city

Let us illustrate with Figure 8.1 how the boundary length of a constituency is derived from the parameters given above. First, the boundaries of all contained base units are summed up. Consider, e.g., the two northeastern base units of the depicted constituency. Since they share a common boundary, this segment is counted twice when we add up the total boundary lengths of all the constituency's base units. However, the parts of each base unit boundary that coincide with the outer constituency boundary are counted only once. Consequently, we obtain the boundary length of the constituency by subtracting each shared boundary twice from the previous sum. In the same manner, we calculate which length of the constituency's boundary corresponds to district boundaries. In the figure, this is the segment of the dashed orange boundary that is additionally marked blue. Dividing this by the constituency's total boundary length renders the district boundary percentage. The result for the example provided by the figure is 0.78 due to the northwestern non-district boundary.

We now express z_p in terms of the other variables as described above:

$$z_p = \frac{\sum\limits_i b_i^{dist} \cdot x_i - \sum\limits_{\{v_i,v_j\} \in E} 2 \cdot b_{ij}^{dist} \cdot y_{ij}}{\sum\limits_i b_i \cdot x_i - \sum\limits_{\{v_i,v_j\} \in E} 2 \cdot b_{ij} \cdot y_{ij}}$$

$$\Leftrightarrow \sum_i b_i^{dist} \cdot x_i - \sum_{\{v_i,v_j\} \in E} 2 \cdot b_{ij}^{dist} \cdot y_{ij} = z_p \left(\sum_i b_i \cdot x_i - \sum_{\{v_i,v_j\} \in E} 2 \cdot b_{ij} \cdot y_{ij} \right)$$

$$\Leftrightarrow \sum_i b_i^{dist} \cdot x_i - \sum_{\{v_i,v_j\} \in E} 2 \cdot b_{ij}^{dist} \cdot y_{ij} = \sum_i b_i \cdot x_i \cdot z_p - \sum_{\{v_i,v_j\} \in E} 2 \cdot b_{ij} \cdot y_{ij} \cdot z_p$$

The last equation still contains products of variables, i.e., $x_i \cdot z_p$ and $y_{ij} \cdot z_p$. This is not allowed in a MIP, but since both products consist of one binary and one continuous

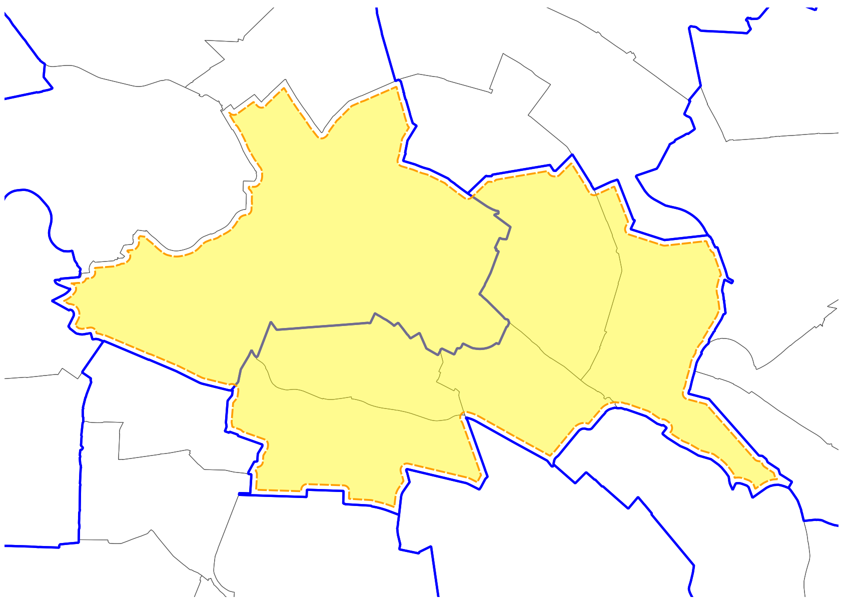


Figure 8.1.: *Administrative boundaries of a constituency (yellow). Thick blue lines mark district boundaries, thin gray ones base unit boundaries. The constituency boundary is the dashed orange line. Source of geometric raw data: Senatsverwaltung für Stadtentwicklung und Umwelt Berlin, CC BY 3.0 DE, `https://creativecommons.org/licenses/by/3.0/de/`*

variable they may be linearized through the introduction of new variables $\tilde{x}_i$ and $\tilde{y}_{ij}$:

$$\begin{aligned} \tilde{x}_i \in [0,1], \tilde{x}_i &= x_i \cdot z_p && \forall v_i \in V \\ \tilde{y}_{ij} \in [0,1], \tilde{y}_{ij} &= y_{ij} \cdot z_p && \forall \{v_i, v_j\} \in E \end{aligned}$$

Employing these new variables the previous equation is linearized:

$$\sum_i b_i^{dist} \cdot x_i - \sum_{\{v_i,v_j\} \in E} 2 \cdot b_{ij}^{dist} \cdot y_{ij} = \sum_i b_i \cdot \tilde{x}_i - \sum_{\{v_i,v_j\} \in E} 2 \cdot b_{ij} \cdot \tilde{y}_{ij}$$

Additional constraints are added to ensure that the variables $\tilde{x}_i$, $\tilde{y}_{ij}$ and z_p adopt suitable values:

$$\begin{aligned} &\tilde{x}_i \leq x_i && \forall v_i \in V && (8.3) \\ &\tilde{x}_i \leq z_p && \forall v_i \in V && (8.4) \\ &\tilde{x}_i \geq z_p + x_i - 1 && \forall v_i \in V && (8.5) \\ &\tilde{x}_i \in [0,1] && \forall v_i \in V \\ &\tilde{y}_{ij} \leq y_{ij} && \forall \{v_i, v_j\} \in E \\ &\tilde{y}_{ij} \leq z_p && \forall \{v_i, v_j\} \in E \\ &\tilde{y}_{ij} \geq z_p + y_{ij} - 1 && \forall \{v_i, v_j\} \in E \\ &\tilde{y}_{ij} \in [0,1] && \forall \{v_i, v_j\} \in E \end{aligned}$$

Due to inequality (8.3), $\tilde{x}_i$ is set to zero if x_i equals zero. Otherwise, $\tilde{x}_i$ equals z_p because of inequalities (8.4) and (8.5). Hence, $\tilde{x}_i$ behaves precisely like the product $x_i \cdot z_p$ and controls the value of z_p as intended. The same holds for $\tilde{y}_{ij}$, likewise. Consequently, z_p measures the adherence to administrative boundaries as a percentage.

In most cases, z_p is exactly the value that z^{bound} should assume. However, there may be districts (or district-free cities) whose population exceeds the constituency population bound, i.e., is greater than $1.25\overline{p}^{const}$. We call such a district *large* and any other district *small*. A large district cannot be covered by a single constituency so that its base units must be assigned to multiple constituencies. Therefore, the non-adherence to administrative boundaries should not be penalized for a constituency that only consists of base units of a single large district. To model this behavior, we first define a subset of nodes V_{aux} and a subset of edges E_{aux}:

$$\begin{aligned} V_{aux} &= \{v_i \in V \mid v_i \text{ belongs to a small district}\} \\ E_{aux} &= \{\{v_i, v_j\} \in E \mid \ v_i \text{ and } v_j \text{ belong to different districts}\} \end{aligned}$$

V_{aux} comprises all nodes that pertain to a small district. E_{aux} contains all edges that correspond to a district boundary. If $V_{aux} = V$, then no large district exists and we define $z^{bound} = z_p$.

Otherwise we introduce a further auxiliary variable z_{aux} as:

$$z_{aux} \in \{0,1\} \text{ with } z_{aux} = \begin{cases} 1 & \text{if not all nodes of the constituency belong to a single} \\ & \text{large district} \\ 0 & \text{otherwise} \end{cases}$$

Additional constraints ensure that z_{aux} assumes the value 1 if the constituency either contains a node from a small district or an edge that corresponds to a district boundary:

$$z_{aux} \geq \sum_{v_i \in V_{aux}} \frac{1}{|V_{aux}|} x_i \tag{8.6}$$

$$z_{aux} \geq y_{ij} \qquad \forall \{v_i, v_j\} \in E_{aux} \tag{8.7}$$

Finally, we define

$$z^{bound} = \max\{z_p, (1 - z_{aux})\} \quad . \tag{8.8}$$

Due to the maximization of the objective function, equation (8.8) can be expressed via the following inequality:

$$z^{bound} \leq z_p + (1 - z_{aux})$$

If the constituency is entirely contained within a large district, then z_{aux} may assume the value 0 and thus z^{bound} equals 1. Otherwise, the constituency must contain a node from a small district or nodes from different large districts. In both cases, z_{aux} is forced to 1 due to inequalities (8.6) and (8.7) and it holds $z^{bound} = z_p$.

Population Balance

The next objective is population balance. Variable $z^{bal} \in [0,1]$ rewards constituencies staying close to the mean population. The European Commission for Democracy through Law (also known as the Venice Commission), an advisory board of the Council of Europe in questions of constitutional law, provides additional guidelines for this

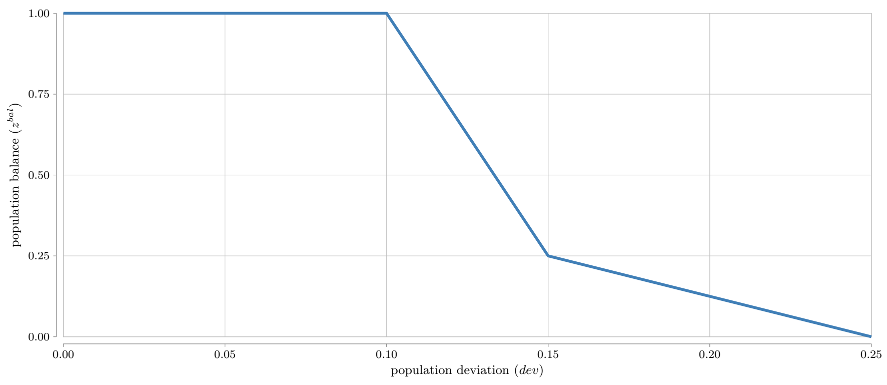


Figure 8.2.: *Piecewise linear function for modeling population balance. Here, the grid points are defined as $f(0) = 1, f(0.1) = 1, f(0.15) = 0.25, f(0.25) = 0$. The function is therefore designed to take into account the advice from the Venice Commission.*

goal [58]. It recommends that "The permissible departure from the norm should not be more than 10%, and should certainly not exceed 15% except in special circumstances (protection of a concentrated minority, sparsely populated administrative entity)."

Let

$$dev = \frac{\sum_i x_i p_i}{\overline{p}^{const}} - 1 \tag{8.9}$$

be the deviation from the mean population as in constraint (8.2). Figure 8.2 illustrates how the population balance z^{bal} is derived from the population deviation dev via a piecewise linear function f:

$$z^{bal} = f(dev)$$

The function f is defined through up to six grid points

$$a_h \in \{0\%, 5\%, 10\%, 15\%, 20\%, 25\%\}$$

with corresponding monotonically decreasing function values $f(a_h)$, where $f(0\%) = 1$ and $f(25\%) = 0$. Denoting the number of segments as n_{seg}, it holds $h \in \{0, \dots, n_{seg}\}$. The Venice Commission's advice is then taken into consideration by choosing suitable a_h and $f(a_h)$. For this thesis, we assume that f corresponds to the function depicted in Figure 8.2.

With the help of additional variables s_h^{seg} and dev_h^{seg}, it is now possible to model z^{bal} through linear constraints:

$$z^{bal} = \sum_{h=1}^{n^{seg}} s_h^{seg} f(a_h) + grad_h(dev_h^{seg} - a_h s_h^{seg}) \tag{8.10}$$

$$|dev| = \sum_{h=1}^{n^{seg}} dev_h^{seg} \tag{8.11}$$

$$\sum_{h=1}^{n^{seg}} s_h^{seg} = 1 \tag{8.12}$$

$$a_{h-1} s_h^{seg} \leq dev_h^{seg} \leq a_h s_h^{seg} \qquad \forall h \in [n^{seg}] \tag{8.13}$$

$$s_h^{seg} \in \{0, 1\} \qquad \forall h \in [n^{seg}] \tag{8.14}$$

Due to equation (8.12) and the integer constraints (8.14), the variables s_h^{seg} can be interpreted to select a single segment. Since s_h^{seg} equals zero for all but the selected segment h', equation (8.11) and the inequalities (8.13) guarantee that $dev_{h'}^{seg}$ corresponds to the constituency's absolute deviation. Let $grad_h$ be the gradient of segment h:

$$grad_h = \frac{f(a_h) - f(a_{h-1})}{a_h - a_{h-1}} \qquad \forall h \in [n^{seg}]$$

Equation (8.10) then ensures that $z^{bal} = f(dev_{h'}) = f(|dev|)$ holds by activating the linear function of segment h'.

With z^{bal} taking on the desired function value, the only remaining issue is that the use of the absolute value function as in equation (8.11) is not permitted in a MIP. Since z^{bal} is maximized and function f monotonically decreasing, it follows that the sum

$$\sum_{h=1}^{n^{seg}} dev_h^{seg}$$

will be minimized. Consequently, it suffices to replace equation (8.11) with the following two inequalities, rendering $|dev|$ as a lower bound for the sum:

$$dev \leq \sum_{h=1}^{n^{seg}} dev_h^{seg}$$
$$-dev \leq \sum_{h=1}^{n^{seg}} dev_h^{seg}$$

Continuity

The last objective is the continuity of constituencies over two successive elections represented by variable $z^{cont} \in [0, 1]$. Let the constituencies of the previous election be given by $\mathcal{C}^{old} = \{C_1^{old}, \ldots, C_{n_c^{old}}^{old}\}$. Continuous variables z_k^{shift} are introduced to determine the population shift of the computed constituency with respect to each previous constituency $C_k^{old} \in \mathcal{C}^{old}$. The population shift is the amount of population that is removed or added to the constituency relative to the previous constituency's total population:

$$z_k^{shift} = \frac{\sum_{v_i \in C_k^{old}} p_i(1 - x_i) + \sum_{v_i \notin C_k^{old}} p_i x_i}{\sum_{v_i \in C_k^{old}} p_i} \qquad \forall k \in [n_c^{old}]$$

Furthermore, we define

$$\overline{z}_k^{shift} = 1 + \frac{1.25\overline{p}^{const}}{\sum_{v_i \in C_k^{old}} p_i} \quad .$$

Based on the constituencies' population limit it follows $z_k^{shift} \in [0, \overline{z}_k^{shift}]$.

In order to select a previous constituency that minimizes population shift, decision variables z_k^{sel} are defined:

$$z_k^{sel} \in \{0, 1\} \text{ with } z_k^{sel} = \begin{cases} 1 & \text{if continuity is measured based on} \\ & \text{previous constituency } C_k^{old} \\ 0 & \text{otherwise} \end{cases} \qquad \forall k \in [n_c^{old}]$$

A constraint ensures that exactly one of the previous constituencies is selected:

$$\sum_{k\in[n_c^{old}]} z_k^{sel} = 1$$

The selected population shift is then given by the term

$$\sum_{k\in[n_c^{old}]} z_k^{sel} z_k^{shift} \in [0, \max_{k\in[n_c^{old}]}\{\overline{z}_k^{shift}\}] \tag{8.15}$$

and the continuity is defined as

$$z^{cont} = \max\{0, 1 - \sum_{k\in[n_c^{old}]} z_k^{sel} z_k^{shift}\} \quad . \tag{8.16}$$

To transform equation (8.16) into a linear constraint, the function max must first be replaced. Therefore, an auxiliary decision variable $z_{max} \in \{0,1\}$ is introduced to define two bounds for z^{cont}:

$$z^{cont} \leq (1 - \sum_{k\in[n_c^{old}]} z_k^{sel} z_k^{shift}) + \max_{k\in[n_c^{old}]}\{\overline{z}_k^{shift}\}(1 - z_{max}) \tag{8.17}$$

$$z^{cont} \leq z_{max} \tag{8.18}$$

For $z_{max} = 0$ the second bound restricts z^{cont} to 0. Additionally, the term

$$\max_{k\in[n_c^{old}]}\{\overline{z}_k^{shift}\}(1 - z_{max})$$

ensures that upper bound (8.17) is non-negative and the problem remains feasible. If $z_{max} = 1$, then the upper bound of z^{cont} is

$$1 - \sum_{k\in[n_c^{old}]} z_k^{sel} z_k^{shift} \quad .$$

Due to the maximization of the objective z^{cont} will always assume the largest possible value and consequently the max function in equation (8.16) is remodeled correctly.

The only remaining task is to linearize the products $z_k^{sel} z_k^{shift}$. This is accomplished via additional variables $\tilde{z}_k^{shift} \in [0, \overline{z}_k^{shift}]$ with the following constraints:

$$\tilde{z}_k^{shift} \leq z_k^{shift} \qquad \forall k \in [n_c^{old}] \tag{8.19}$$

$$\tilde{z}_k^{shift} \leq \overline{z}_k^{shift} z_k^{sel} \qquad \forall k \in [n_c^{old}] \tag{8.20}$$

$$\tilde{z}_k^{shift} \geq \overline{z}_k^{shift} z_k^{sel} + z_k^{shift} - \overline{z}_k^{shift} \qquad \forall k \in [n_c^{old}]$$

Since the maximization of continuity corresponds to the minimization of population shift it suffices to define lower bounds for the linearization variables $\tilde{z}_k^{shift}$. Consequently, the upper bounds (8.19) and (8.20) may be discarded. Using the linearization variables, the selected population shift (8.15) can now be expressed as a linear constraint:

$$z^{shift} = \sum_{k\in[n_c^{old}]} \tilde{z}_k^{shift}$$

Combined Objective

Since the fulfillment of each of the four different goals is measured by a different variable, the objective function can be defined as the weighted sum of these metrics:

$$\max \sum_{o \in Obj} c^o z^o$$

Here, $Obj = \{tol, bound, bal, cont\}$ is the set of objectives and for each objective o holds $c^o \geq 0$. If there is some objective $o \in Obj$ with $c^o = 0$, then all corresponding auxiliary variables and constraints that are not employed elsewhere may be removed from the model.

8.1.2. Custom Constraints

As already stated in the beginning of this chapter, constituencies must obey additional constraints apart from the ones that are implicitly required for the definition of the objective function.

Firstly, constituencies are not to cross state boundaries. As already described before, the number of constituencies of a state can easily be precomputed given its population so that districting can be performed separately for each state.

Secondly, the constituencies' population is not allowed to deviate from the mean population by more than 25% which may be guaranteed through the following constraint:

$$-0.25 \leq \frac{\sum_i x_i p_i}{\overline{p}^{const}} - 1 \leq 0.25 \tag{8.21}$$

As mentioned in Section 8.1.1, this is already implicitly ensured by constraint (8.2). However, if that constraint is excluded because population tolerance is not considered as an objective, i.e., $c^{tol} = 0$, then the population bounds (8.21) are required.

8.2. Framework Plug-Ins

We implemented two additional problem-specific plug-ins for solving GPDP instances, an initializer and a heuristic pricer. Either plug-in employs the greedy approach of Algorithm 8.1 to determine feasible clusters to add to the RMP.

First a root node is selected from some initialization cluster C_{init} as the basis for a new cluster C. Cluster C is then stepwise extended by adding a neighbor of some node that already belongs to C. The best candidate node to add is selected according to a cost function. As long as C does not yet contain all nodes from the initialization cluster, only these missing nodes $C_{init} \setminus C$ may be added. When the population of cluster C is larger than the minimum constituency population pop_{min} derived from constraint (8.21), a copy is inserted into the set of clusters to add to the RMP. When cluster C can no longer be extended without exceeding the maximum constituency population pop_{max}, the algorithm terminates by returning the set of feasible clusters.

The difference between the initializer and the pricer plug-in is the definition of the initial cluster C_{init}, the root node selection and the cost function for the candidate nodes. Note that Algorithm 8.1 is executed repeatedly for different initialization clusters.

Algorithm 8.1 Greedy Constituencies

```
function computeConstituencies($G, C_{init}, pops, pop_{min}, pop_{max}$)
    $constituencies \leftarrow \emptyset$
    $C \leftarrow \emptyset$
    $pop_C \leftarrow 0$
    $bestCandidate \leftarrow$ null
    $root \leftarrow$ computeRootNode($C_{init}$)
    if $pops[root] \leq pop_{max}$ then
        $bestCandidate \leftarrow root$
    while $bestCandidate \neq$ null do
        $C \leftarrow C \cup \{bestCandidate\}$
        $pop_C \leftarrow pop_C + pops[bestCandidate]$
        if $pop_C \geq pop_{min}$ then
            $constituencies \leftarrow constituencies \cup \{C\}$
        $bestCost \leftarrow \infty$
        $bestCandidate \leftarrow$ null
        for $u \in C$ do
            for $\{u, v\} \in E$ do
                if $v \notin C \wedge pop_C + pops[v] \leq pop_{max}$ then
                    if $(|C| < |C_{init}| \wedge v \in C_{init}) \vee |C| \geq |C_{init}|$ then
                        $cost \leftarrow$ computeCandidateCost($u, v, C$)
                        if $cost < bestCost$ then
                            $bestCost \leftarrow cost$
                            $bestCandidate \leftarrow v$
    return $constituencies$
```

Moreover, the framework implementation is differs slightly by maintaining a map of candidate nodes and costs for increased efficiency.

The *Greedy Initializer (GI)* is given a partitioning to perform the algorithm on each of its clusters. This allows, e.g., to obtain clusters from the constituencies of the previous election. Even if some constituencies are no longer feasible due to population changes, it may be possible to derive similar constituencies that are somewhat smaller or larger. The cost function for the candidate nodes takes int account three different factors: whether u and v belonged to the same constituency in the previous election, whether u and v belong to the same district and the population deviation of $C \cup \{v\}$ from the mean constituency population. Note that these are determining factors for the objective of the PP. Thus, the cost function attempts to create clusters with a high objective function coefficient in the RMP. In order to select a root node, we first define the root district d_{root} as the district whose population is covered by C_{init} to the largest percentage:

$$d_{root} = \arg\max_{d \in D}\{\frac{\sum_{v \in C_{init} \cap d} pop[v]}{\sum_{v \in d} pop[v]}\}$$

Here, D is a partitioning of V into administrative districts. If multiple districts maximize the term, we choose one of maximal population. The root node is the node of maximum population of the root district. The root district is chosen with the intention of reducing the need to cross administrative boundaries.

The heuristic *Greedy Pricer (GP)* executes Algorithm 8.1 once for each node $v \in V$ with $\{v\}$ as the initial cluster. Consequently, v is also the root node. The candidate costs are defined similar to the initializer. However, instead of the population deviation, the pricer considers the dual value of u relative to the corresponding population. This way, it heuristically maximizes the objective of the PP (5.5).

9. Computational Results

We evaluated the implemented framework by applying different combinations of pricers and initializers to numerous OCPP and GPDP instances.

For the execution of our experiments we employed HTCondor[1], a batch system for the workload management of compute-intensive jobs. Although HTCondor allows for parallel computation, we employed it only as a scheduler for running serial jobs on a cluster of identical machines. The employed cluster called *clustOR* is maintained by the Chair of Operations Research of RWTH Aachen University. Table 9.1 lists the machines' hardware specification. The operating system is a modified version of Debian 9 (Stretch).

Table 9.1.: *Hardware specification of the clustOR machines.*

Component	**Specification**
Model	HP ProLiant SE316M1
Processor (CPU)	Xeon L5630 Quad Core 2.13 GHz
Memory (RAM)	16 GB DDR3 RAM
Hard Disk	2x 146 GB SAS 10K
Power	Dual Redundant 400W Power Supplies (PSU)

In the following section, we introduce performance profiles as a tool for comparing the performance of different solving algorithms. Additionally, we state the performance metrics used in the evaluation. We then proceed by describing the datasets used for our experiments and examining the computational results of different plug-in combinations. Finally, we briefly summarize our most relevant findings.

9.1. Performance Metrics and Performance Profiles

Let $\mathcal{P}$ be a set of CVCP instances and $\mathcal{A}$ a set of solving algorithms. We measure the performance of an algorithm $a \in \mathcal{A}$ on instance $p \in \mathcal{P}$ by different performance metrics. Table 9.2 provides an overview over these metrics, which we describe in the following.

The first metric is the solving time t in seconds. Note that we set a time limit t_{max}, i.e., the execution of an algorithm is stopped at time t_{max} even if it has not yet determined an optimal solution. Let $t'_{a,p}$ be the execution time of algorithm a on instance p. Then we define:

$$t_{a,p} = \begin{cases} t'_{a,p} & \text{if } a \text{ determined an optimal solution on } p \text{ within time } t_{max} \\ \infty & \text{otherwise} \end{cases}$$

[1]`http://htcondor.org/`, (last accessed: 08/19/2018)

The pricing round (PR) time t^{PR} indicates the average execution time of a single pricing round:

$$t^{PR}_{a,p} = \frac{\text{pricing time of a on p}}{\text{number of pricing rounds of a on p}}$$

In each feasible node of the branch-and-bound tree, the pricing loop is executed as long as cluster variables of positive reduced costs are found that can be added to the RMP. Thus, the last PR of each pricing loop is the one where no such variables can be determined and the computed solution of the LPR is optimal over all MP variables. The unsuccessful PR time t^{UPR} is the average execution time of all PRs where no variable is inserted into the RMP:

$$t^{UPR}_{a,p} = \frac{\text{sum of unsuccessful pricing round times of a on p}}{\text{number of unsuccessful pricing rounds of a on p}}$$

Note that unsuccessful PRs do not exclusively occur at the end of the pricing loops, but on rare occasions also due to pricer calls by heuristics of the SCIP framework.

The gap g measures the relative difference between primal bound pb and dual bound db:

$$g_{a,p} = \begin{cases} \left|\frac{pb_{a,p}-db_{a,p}}{pb_{a,p}}\right| & \text{if } pb_{a,p} \notin \{0, -\infty\} \\ 0 & \text{if } pb_{a,p} = 0 \text{ and } db_{a,p} = 0 \\ \infty & \text{otherwise} \end{cases}$$

It holds $g_{a,p} = 0$ iff an optimal solution is found within time t_{max}. Moreover, both pb and db may be infinite.

Table 9.2.: *Performance metrics.*

Symbol	Value Range	Name
t	$(0,\infty) \cup \{\infty\}$	solving time
t^{PR}	$(0,\infty) \cup \{\infty\}$	PR time
t^{UPR}	$(0,\infty) \cup \{\infty\}$	unsuccessful PR time
g	$[0,\infty) \cup \{\infty\}$	gap

Based on the introduced performance metrics, we define performance profiles [21]. These are designed particularly to ease the comparison of the performance of different solving algorithms on a set of problem instances. Let $m \in (0,\infty) \cup \{\infty\}$ be some performance metric where a lower score indicates a higher performance. A suitable metric is, e.g., the solving time t. The corresponding performance ratio is defined as the performance $m_{a,p}$ of a on p relative to the best performance $m_{a',p}$ on p amongst all algorithms $a' \in \mathcal{A}$:

$$r^m_{a,p} = \begin{cases} \infty & \text{if } \min_{a' \in \mathcal{A}}\{m_{a',p}\} = \infty \\ \frac{m_{a,p}}{\min_{a' \in \mathcal{A}}\{m_{a',p}\}} & \text{otherwise} \end{cases} \in [1,\infty) \cup \{\infty\}$$

Based on the performance ratio, we define the performance ratio profile ρ^m_a of solver a as the following function:

$$\rho^m_a \quad : \quad [1,\infty) \cup \{\infty\} \to [0,1], \quad \mu \mapsto \frac{1}{|\mathcal{P}|}|\{p \in \mathcal{P} : r^m_{a,p} \leq \mu\}|$$

Hence, $\rho_a^m(\mu)$ is the probability that the performance of solver a is at most μ times worse than the best performance. Moreover, ρ_a^m is the cumulative distribution function of the performance ratio. The higher the values of $\rho_a^m(\mu)$, the better the algorithm performs with regard to metric m.

The performance ratio profile requires all metric values to be greater than zero. However, some metrics like the gap include values equal to or smaller than zero. In analogy to the performance ratio profile, we define a performance delta profile for such metrics. Hence, let $m \in (-\infty, \infty) \cup \{\infty\}$ again be a performance metric where lower scores represent higher performance. We define the performance delta as the difference between an algorithm's performance $m_{a,p}$ and the best performance $m_{a',p}$ amongst all solvers:

$$\delta_{a,p}^m = \begin{cases} \infty & \text{if } \min_{a' \in \mathcal{A}}\{m_{a',p}\} = \infty \\ m_{a,p} - \min_{a' \in \mathcal{A}}\{m_{a',p}\} & \text{otherwise} \end{cases} \quad \in [0, \infty) \cup \{\infty\}$$

In the same manner as before, we define the performance delta profile ϕ_a^m of solver a as:

$$\phi_a^m \quad : \quad [0,\infty) \cup \{\infty\} \to [0,1], \quad \Delta \mapsto \frac{1}{|\mathcal{P}|}|\{p \in \mathcal{P} : \delta_{a,p}^m \leq \Delta\}|$$

Thus, $\phi_a^m(\Delta)$ is the probability that the performance of solver a is worse than the best performance by a difference of at most Δ.

To be suitable for a performance delta profile, the values of a metric should be independent of the instance size. Otherwise, the comparison of the performance deltas $\delta_{a,p}^m$ over different instances $p \in \mathcal{P}$ is not meaningful. The solving time is therefore an unsuitable metric. In contrast, metrics that are bounded to a closed interval are typically well suited.

For the comparison of different solvers on CVCP instances, we employ performance ratio profiles ρ_a^t, $\rho_a^{t^{PR}}$ and $\rho_a^{t^{UPR}}$ as well as the performance delta profile ϕ_a^g.

9.2. Odd Cycle Packing Problem

In the following, we analyze the performance of the SIP, the SPSP and the OCPP-specific plug-ins on numerous instances.

9.2.1. Dataset

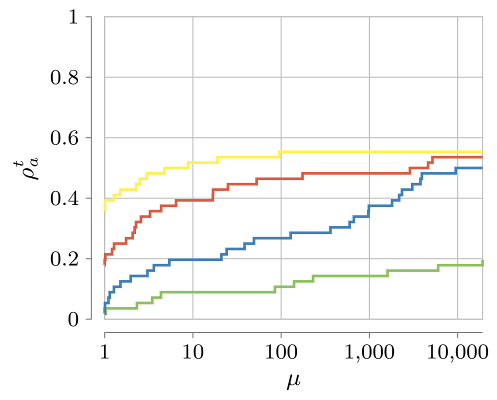
To evaluate the CVC Framework on the OCPP, we used a collection of graph coloring instances[1] related to the Implementation Challenges of the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS)[2]. The collection comprises diverse graphs of different origins in two different formats. We only used graphs of the non-binary format *.col* and excluded three of them from the evaluation. The first graph *latin_square_10* was excluded because with 900 nodes and 307,350 edges it is by far the largest. The second instance *myciel2* was not formatted correctly. Finally, we did not consider the graph *david* in the evaluation because the solving process resulted in a

[1]`https://mat.gsia.cmu.edu/COLOR/instances.html` (last accessed: 08/19/2018)
[2]`http://dimacs.rutgers.edu/archive/Challenges/` (last accessed: 08/19/2018)

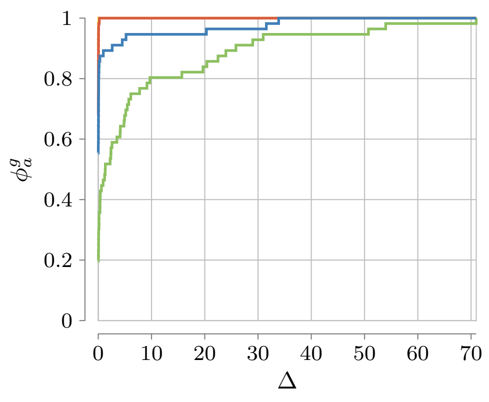
runtime error for one experimental setup due to a bug in the employed SCIP library. Table B.1 states different characteristics like the number of nodes and edges of the remaining 56 instances. Note that all loops were removed from the original graphs.

9.2.2. Evaluation

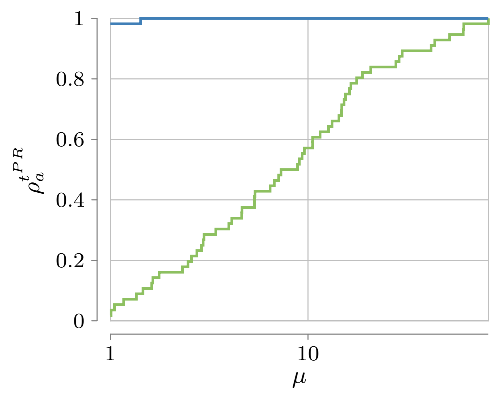
All OCPP instances were solved with a time limit of 2 hours. We evaluated four different experimental setups: the SIP alone, the SPSP alone, the SIP combined with the heuristic TCP and the SIP after initialization by the TCI. Section 6.4.1 already stated that the SPSP is also a heuristic pricer because there are certain connected clusters which it cannot compute. However, we treated it as an exact pricer here in order to compare its efficiency and effectiveness to the SIP. I.e., if the SPSP did not find any cluster of positive reduced costs, we assumed that such a cluster does not exist and terminated the pricing loop. In consequence, a solution computed by the SPSP setup is not necessarily optimal, even if the solver terminates with gap 0. However, we derived from the raw data that all solutions where the SPSP setup reached gap 0 are indeed optimal.



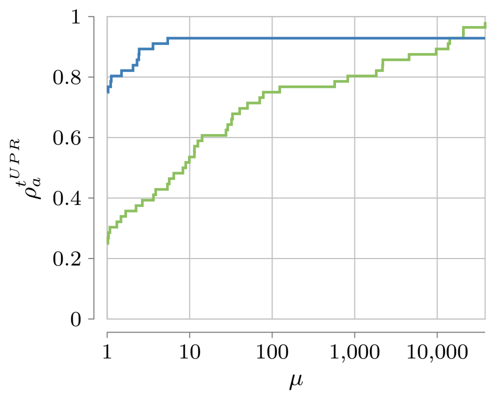
(a) *solving time*



(b) *gap*



(c) *PR time*



(d) *unsuccessful PR time*

Figure 9.1.: *OCPP performance profiles.*

Figure 9.1 shows the resulting performance profiles of all solvers. The underlying metric values are listed in Tables B.2–B.4 in the appendices. Note that the axis for μ in the performance ratio plots is logarithmic. Moreover, we based the performance profiles for PR time and unsuccessful PR time only on the single-pricer setups.

Let us begin by comparing the two single-pricer setups. The solving time performance profile indicates that the SIP setup is more efficient than the SPSP setup. The PR time performance profiles of the two pricers clearly show that the SPSP requires on average significantly more time for solving a single PP. On 42.86% of the instances, an average SPSP execution takes at least 10 times as long as an average SIP execution. Recall from Section 6.4.1 that the SPSP solves multiple MIPs, each corresponding to some center node $v_c \in V$ of the instance graph. In an unsuccessful PR, the SPSP has to solve a subproblem for all the nodes $v_c \in V$. The unsuccessful PR time performance profiles show that this property has a particularly strong impact on the pricing efficiency. For 23.21% of the instances, the SPSP takes on average at least 100 times longer than the SIP for solving a PP where no variable has positive reduced costs. Another aspect is that the SPSP does not obtain an upper bound for the reduced costs when the subproblem is not solved for all nodes $v_c \in V$. Then it is not possible to derive the upper bound (5.23) for the objective in the current node of the branch-and-bound tree. Consequently, solving may additionally consume more time because of weaker local bounds. Due to the long execution times, the SPSP setup managed to solve only 19.64% of the instances to optimality within the time limit compared to 50.00% for the SIP setup. The higher efficiency also results in a better gap performance by the SIP setup.

Based on the solving time and gap we see that the SIP+TCI setup performed best overall and determined optimal solutions for the largest amount of instances (55.36%). Note that the execution time of the TCI is nearly insignificant with respect to the total solving time. The raw data shows that the highest initialization duration is 1.17s. For 78.57% of the instances, the initialization even lasted no more than 0.1s. The high efficiency of the SIP+TCI setup indicates that three-cycles are often crucial for obtaining LP solutions of high objective. Indeed, the cardinality of 54.84% of the optimal packings computed by the setup SIP+TCI equals the upper bound $k_{max} = \lfloor \frac{n}{3} \rfloor$ from Chapter 7. Thus, any optimal solution of these instances comprises three-cycles. We derive that the initial insertion of three-cycles typically reduces the time required for pricing and may even render pricing completely obsolete on some instances.

Finally, we examine the SIP+TCP setup. Section 7.2 explained that the heuristic TCP inserts a fixed percentage of the three-cycles with positive reduced costs into the RMP. Three-cycles are only computed once and cycles already inserted into the RMP do no longer need to be checked. If now the percentage of inserted cycles is 100%, the TCP inserts all three-cycles into the RMP on its first call and therefore behaves similar to the TCI. If the percentage of inserted cycles is 0%, then the TCP does not insert any variables and the SIP+TCP setup is equivalent to the SIP setup. We set the percentage of inserted cycles to 10%. Assuming that the execution of the TCP is fast compared to the SIP, it was to be expected that the SIP+TCP setup performs somewhere in between the SIP+TCI and the SIP setup. With respect to the solving time, this is exactly the case. Also regarding the number of instances solved to optimality, the SIP+TCP setup ranks second with 53.57%. However, its gap performance is even nearly identical to the

SIP+TCI setup. With regard to solving time, the raw data shows that the SIP+TCI setup outperforms the SIP+TCP setup the most on instances where the former never enters the pricing loop. However, the TCP still inserts the most promising three cycles into the RMP within only a few pricer calls. Since the number of variables in the RMP is larger for the SIP+TCI setup, solving the RMP additionally requires more time. This provides a trade-off for the repeated execution of the TCP. Consequently, the TCI thus loses its advantage over the TCP on instances which take longer to solve. For all instances with gap greater zero the solving process was executed for two hours. This explains why there is no considerable difference between the two setups regarding gap performance.

9.3. German Political Districting Problem

As for the OCPP, we examine the performance of the CVC Framework's general puropose pricers as well as the problem-specific plug-ins.

9.3.1. Dataset

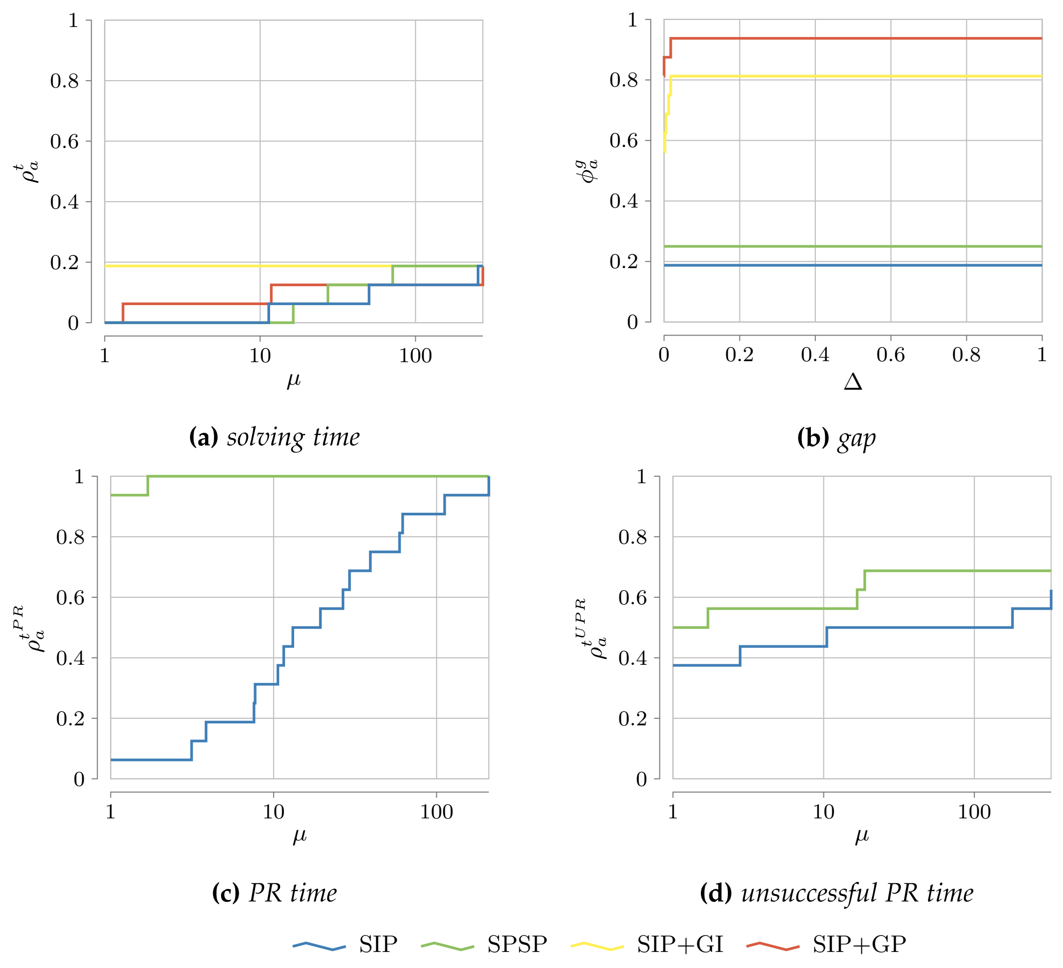
The GPDP instances are based on version 1.0 of the dataset *GeoBevDE*. This dataset is a collection of German geometric and population data. It was created by S. Goderbauer and M. Wicke in 2016 at the Chair of Operations Research at RWTH Aachen University. The data is based on multiple sources including maps provided by the German Federal Returning Officer, OpenStreetMap[1] (Copyright by OpenStreetMap contributors), and geometric and population data from the administrative offices of the largest German cities. From the dataset we created one GPDP instance for each German federal state, a total of 16. Table C.1 in the appendices lists each instance with the corresponding number of nodes and edges.

9.3.2. Evaluation

For the GPDP instances, we set the time limit to 3 hours. The objective function coefficients of the PP's metric variables were defined as $c^{tol} = 0$ for population tolerance, $c^{bound} = 0.3$ for boundary adherence, $c^{bal} = 0.1$ for population balance and $c^{cont} = 0.6$ for continuity. Similar to the OCPP, we evaluated four different setups: both the SIP and the SPSP alone, the SIP with the GI and the SIP with the GP.

The behavior of the SIP and the SPSP were slightly modified in comparison to the OCPP evaluation. To speed up the execution, the pricers terminate as soon as they have found twice a new best solution. The corresponding cluster variables are then added into the RMP as usual, even if neither one is optimal. In order to count only solutions of positive reduced costs, we require all feasible clusters to have an objective value of at least 0.001. Note that the solution process is therefore no longer exact because clusters with smaller positive reduced are no longer considered. Let n_c^* be the fixed number of constituencies as in Chapter 8. If the solver terminates with gap 0, the upper bound (5.23) still guarantees that the objective value of the computed solution is at most 0.001 n_c^* lower than that of an optimal one. For simplicity, we thus still refer to the resulting

[1]`www.openstreetmap.org/copyright` (last accessed: 08/19/2018)



(a) *solving time* **(b)** *gap* **(c)** *PR time* **(d)** *unsuccessful PR time*

Figure 9.2.: *GPDP performance profiles.*

solution as being optimal. As for the OCPP, we verified again that the objective values of all solutions computed by the heuristic SPSP with gap 0 equal the objective values of the corresponding solutions of the exact SIP.

Figure 9.2 shows the solvers' performance profiles for the GPDP instances. The corresponding metric values are given in Tables C.2–C.4 in the appendices. Due to the smaller number of instances, the results are less clear than in the OCPP evaluation. This holds particularly for the solving time and the gap because only a low percentage of instances was solved to optimality and for many instances the solvers could not even determine a feasible primal solution. Again, we based the PP time performance profiles only on the single-pricer setups.

As before, we begin with a comparison of the SPSP and the SIP. Regarding the solving time performance profiles, the SPSP setup slightly outperforms the SIP setup. However, the difference is too small to draw a conclusion on such a small number of instances. The same holds for the number of instances solved to optimality and gap performance. Both setups computed optimal solutions for 18.75% of the instances. The SPSP setup is again somewhat better regarding gap performance because it managed to determine a feasible primal solution on a single instance where the SIP setup did not.

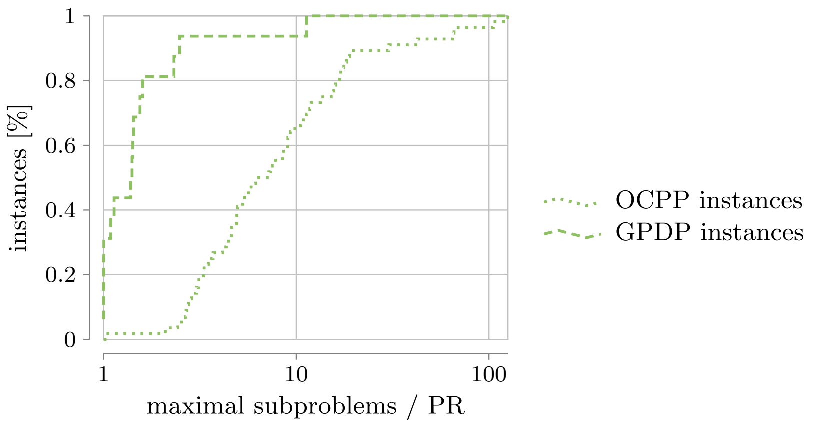


Figure 9.3.: *Average number of subproblems solved by the SPSP per PR.*

In contrast to the OCPP, the SPSP clearly outperforms the SIP regarding PR time performance. Figure 9.3 partially explains this contradictory result. Similar to a performance profile, the x-axis of the diagram represents the average number of subproblems that the SPSP solved per PR. The corresponding y-values indicate for which percentage of instances the given average was not exceeded. The figure shows that the pricer solved significantly less subproblems per PR for the GPDP instances than for the OCPP instances.

Some explanations for this phenomenon can be derived from the raw data. For 75.00% of the instances, neither pricer manages to solve the root node LP within the time limit. Consequently, the first pricing loop is never left. As stated in the previous section, one reason for the inefficiency of the SPSP is having to solve a different MIP for each center node $v_c \in V$ in the last PR of a pricing loop. Since the time limit prevents the SPSP setup on numerous instances from ever reaching such a PR, the measured performance increases. Furthermore, the instances where the setup does manage to solve the root LP are the smallest ones. Hence, even then solving one MIP per node impacts the execution time than usual.

However, the unsuccessful PR time performance profiles show that the SPSP still outperforms the SIP when it has to solve subproblems for all nodes. Even under consideration of the time limit's favorable impact, the SPSP seems therefore more suitable for the GPDP. Nevertheless, the pricer remains heuristic. If optimality must be guaranteed, it may therefore be recommendable to combine the SPSP with the SIP or another exact pricer.

Next, we examine the impact of the GI. Similar to the OCPP, the initializer improves both the solving time and the gap performance. One reason for this is the definition of the weights of the metric variables z^o. The continuity objective measures in how far the computed constituencies match those of the previous election. The GI uses exactly the last elections' constituencies to derive related ones that are still feasible with regard to the new population data. Setting $c^{cont} = 0.6$ prioritizes continuity as an objective and therefore benefits the performance of the GI. We observed that the three optimal solutions computed by the setup are entirely based on the variables created by the initializer. Due to the early insertion of all relevant variables, the SIP+GI setup achieves the shortest solving times amongst all solvers. Moreover, the gap performance indicates that the variables created by the GI promote the derivation of feasible primal solutions also when a problem is not solved to optimality. In contrast to the low percentages of the single-pricer setups, the SIP+GI setup thus managed to determine a finite gap for 81.25% of the instances. The raw data shows additionally that a gap below 10% was obtained for 68.75% of the instances.

The supplementary GP also enhanced the performance of the SIP. Like all other setups, the SIP+GP setup solved 18.75% of the instances to optimality. Despite the limited data available, it seems clear that the GP reduces solving time. Moreover, the GP had a significant impact on the computation of feasible solutions. The SIP+GP setup outperformed all others regarding gap performance by computing a finite gap for 93.75% of the instances. A gap below 10% was accomplished for 87.50% of the instances. In conclusion, the GP outperforms the GI on the larger instances. At least when continuity is prioritized, the initializer is still faster on smaller instances.

9.4. Summary

Summing up the findings from the experiments, we can say first of all that the GPDP is harder to solve than the OCPP. This may be partially due to the PP which comprises more variables and constraints. However, the main reason is that the MP is a partitioning problem. Therefore, the clustering constraints (4.21) are more restrictive and it is harder to compute feasible solutions. While the GPDP requires nodes to be part of exactly one cluster, they are not forced to belong to a cluster in the OCPP. The complexity of finding feasible solutions for the GPDP is illustrated by the percentage of pricing time that each pricer spends on Farkas pricing rather than reduced cost pricing. I.e., this indicates how much of the execution time is spent on obtaining feasibility rather than optimality. For the OCPP, the SIP spent less then 1% of its execution time on Farkas pricing for 89.29% of the instances. For the SPSP, it was even 98.21%. In contrast for the GPDP, the SIP spent all its execution time on Farkas pricing for 75.00% of the instances. The SPSP performed Farkas pricing alone on 31.25% of the instances.

With regard to the general purpose pricers, we determined that the heuristic SPSP setup computed optimal solutions whenever it terminated within the time limit. Nevertheless, the SIP is more efficient than the SPSP on OCPP instances. This behavior is reversed for the GPDP where the SIP keeps only the advantage of being exact. Although we identified certain aspects favoring the SPSP in the GPDP evaluation, we did not completely resolve why either pricer is more fitting for one problem than the other.

The evaluation of additional problem-specific plug-ins showed that initializers and supplementary heuristic pricers provide significant boosts in efficiency. Regarding the OCPP, we determined that the behavior of the TCP can be tuned to be more like the TCI or more passive. In the first case, lots of variables are added to the RMP early on in the solving process. This may help to reduce the number of PRs, but can also increase the solving time of the RMP. For the GPDP, we noted that the GI benefits from the weighting of the continuity constraint. Nevertheless, the supplementary GP achieves achieves higher efficiency on larger instances.

10. Conclusion and Outlook

Motivated by the definition of constituencies for the German federal elections, the main goal of this thesis was the development of a solving approach for customizable graph clustering problems. The computed clustering should be optimal with regard to a user-defined objective and all clusters must be connected and satisfy further application-specific constraints.

We formalized this problem as a MIP in Chapter 4. This so-called CVCP may be employed to compute three different types of clusterings: packings, partitionings and coverings. The problem's NP-hardness was shown by reducing the Set Partitioning Problem to the CVCP. In order to circumvent the issue of symmetry, we proceeded by transforming the original problem formulation into an aggregated extended formulation via discretization-based Dantzig-Wolfe decomposition.

Chapter 5 introduced the concept of branch-and-price for solving a MIP on only a subset of its variables. From the duals of the RMP's LPR, we derived the reduced costs of arbitrary MP variables. This allowed to define the PP based on the CVCP's original formulation. The PP was then used to generate new RMP columns that help to improve the objective function value. To obtain integer solutions, the column generation procedure was integrated into a branch-and-bound approach. Additionally, we explained how to employ Farkas pricing for coping with RMP infeasibility and how dual bounds of the PP translate into upper bounds for the MP.

Aside from presenting the general notion of branch-and-price, we discussed why standard variable branching is unsuitable for the CVCP and elaborated a more fitting branching rule on the basis of Ryan-Foster branching. In particular, we generalized the branching constraints of the aforementioned strategy for partitioning problems to the packing and covering scenario of the CVCP. Additionally, we outlined how these constraints impact the objective function and the space of feasible solutions of the PP.

On the foundation of the SCIP library for mixed integer programming and the LEMON graph library, we implemented our method as a C++ framework for solving arbitrary CVCP instances. Chapter 6 described implementation details like the CVC Framework architecture and the core classes of the solving process. The framework allows the integration of custom plug-ins, e.g., to add problem-specific pricers that exploit special characteristics of some CVCP variant. For covering scenarios and the use of pricers without a unique cluster encoding, the CVC framework implements multiple strategies for handling the generation of duplicate variables. Furthermore, it offers visualizations of graphs and graph clusterings as well as analytic features to examine different aspects of the solving process.

In order to solve PPs, the CVC Framework features two general purpose pricers, the SIP and the SPSP. Both pricers are MIP-based, but they follow different approaches for modeling connectivity through linear constraints. The SIP derives connectivity constraints from node separators and employs branch-and-cut, i.e., lazy constraint insertion, to omit unnecessary constraints in the PP. For the SPSP, the PP consists of one subproblem for each center node $v_c \in V$ of the instance graph. The connectivity constraints of a given subproblem are then based on shortest path trees with root v_c. In contrast to the SIP, the SPSP is heuristic because its connectivity constraints declare certain connected clusters infeasible.

Chapters 7 and 8 introduced the OCPP and the GPDP as specializations of the CVCP. The goal of the OCPP is the computation of the OCPN, i.e., the maximum cardinality of a packing of odd cycles. We showed that it suffices to consider only node-induced cycles for this purpose. The GPDP models the political districting for the German federal elections under consideration of different objectives dictated by German law and European guidelines. For both problems, we integrated combinatorial application-specific plug-ins into the CVC Framework.

Chapter 9 finally presented the computational results of the application of the CVC Framework to a variety of OCPP and GPDP instances. For each problem, we analyzed different experimental setups using performance profiles. Traditional performance (ratio) profiles were employed to evaluate solving time, PR time and unsuccessful PR time. In addition, we defined performance delta profiles as a means of comparing the gap performance of multiple solvers on a larger set of instances. To ensure comparability of the execution times, we ran all experiments on a cluster of identical machines.

Despite it being heuristic, the SPSP was treated as an exact pricer to compare its performance to the SIP. We established that the major weakness of the SPSP is having to solve one subproblem for each center node. This drawback is mitigated by terminating the pricer after the first subproblem that produced a variable of positive reduced costs. However, in the last PR of a pricing loop no such variable exists and solving all subproblems is unavoidable. The SIP clearly outperformed the SPSP in the OCPP evaluation, but the latter proved faster for the GPDP. Partially, this was due to the SPSP solving less subproblems per PR and the time limit favoring the SPSP in the GPDP evaluation.

Finally, the supplementary combinatorial plug-ins were tested in combination with the exact SIP. All of them had a strong positive impact on the solving time and gap performance.

10.1. Outlook

From the results of this thesis we derive numerous subjects for future work and research. Some are related directly to the CVC Framework. For others, the framework may be used as an tool to facilitate further investigation.

First, there are still several open research questions regarding the SPSP. A deeper analysis may help to characterize CVCP variants where the pricer is particularly efficient. Even when the SPSP has to solve too many subproblems per PR to be efficient as an exact pricer, it may additionally still be employed as a heuristic one. In this case it suffices for the SPSP to solve a single subproblem. If no variable of positive reduced costs is found, the PP will be solved by an exact pricer. Different strategies for selecting a subproblem with a suitable center node v_c can be tested. E.g., one can iterate over the nodes, choose one at random or select nodes according to the duals of the clustering constraints. At last, the SPSP also poses some theoretical questions. We proved through a counterexample that the pricer's constraints are sufficient, but not necessary for connectivity. It would be interesting to also investigate and characterize the class of graphs where the connectivity constraints are both sufficient and necessary since the pricer is then exact.

The SCIP library offers a plethora of parameters for controlling the behavior of the branch-and-price process. Adjustments may have positive impact on certain CVCP variants or even CVCPs in general. For the GPDP, we already adapted the MIP-based pricers to only take into account clusters that exceed specified reduced costs and to terminate pricing after two improving solutions have been determined. In addition, one might, e.g., deactivate certain pricers after solving the root node LP or reaching a certain depth in the branch-and-bound tree. The implemented GPDP and OCPP plug-ins can further be fine-tuned by experimenting with different configurations of the algorithms' parameters. E.g., varying the insertion percentage of the TCI may yield a better trade-off between redundant variables in the RMP and the number of PRs. The GPDP-specific plug-ins can be adapted by redefining the candidate costs of the greedy algorithm. For the GP in particular, it is also possible to employ different initialization clusters. Other single-node clusters could be based on the strategies already suggested for the center node selection of the SPSP.

Furthermore, the plug-in architecture of the CVC Framework simplifies the development of new components and their integration into the framework. In particular for the GPDP, the performance of the solving process may still be improved. Additional combinatorial pricers and initializers may provide means for accomplishing better results. On top of that, we did not employ any start heuristics so far. The implemented initializers serve a similar purpose by adding variables into the RMP, but the provision of feasible primal solutions may help to speed up the execution.

Aside from problem-specific plug-ins, one might also want to investigate other techniques that are applicable to CVCPs in general. In Section 3.3.1 we already presented a third approach based on node cuts for defining linear connectivity constraints. Since the concept is not yet implemented, a corresponding pricer would be an ideal extension of the CVC Framework. A completely different strategy is to reduce the size of the PPs based on node coverings of the graph [40]. I.e., each PP only considers clusters that are contained within a given subgraph. Consequently, all nodes and edges that are not in the subgraph can be ignored and the number of variables and constraints decreases. A new subgraph is selected to restrict the next PP.

Moreover, the CVC Framework eases the investigation of other CVCP variants. With the OCPP and the GPDP, we have so far evaluated a packing and a partitioning problem. A detailed analysis of the framework's application to a covering problem is still missing. Finally, some supplementary features of the CVC Framework can be developed further to facilitate research even more. Compatibility with a wider range of file formats helps to apply the framework to new problems and instances. Providing additional data on the solving process and more advanced analytical tools assists the evaluation and comparison of different solving approaches.

Bibliography

[1] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, jan 2005. ISSN 01676377. doi: 10.1016/j.orl.2004.04.002. URL `http://linkinghub.elsevier.com/retrieve/pii/S0167637704000501`. Accessed: 08/19/2018.

[2] Charu C. Aggarwal and Haixun Wang. A Survey of Clustering Algorithms for Graph Data. In *Managing and Mining Graph Data*, pages 275–301. Springer, Boston, MA, 2010. doi: 10.1007/978-1-4419-6045-0_9. URL `http://link.springer.com/10.1007/978-1-4419-6045-0_9`. Accessed: 08/19/2018.

[3] Egon Balas and Manfred W. Padberg. Set Partitioning: A survey. *SIAM Review*, 18(4):710–760, oct 1976. ISSN 0036-1445. doi: 10.1137/1018115. URL `http://epubs.siam.org/doi/10.1137/1018115`. Accessed: 08/19/2018.

[4] Arindam Banerjee and Joydeep Ghosh. Scalable Clustering Algorithms with Balancing Constraints. *Data Mining and Knowledge Discovery*, 13(3):365–395, sep 2006. ISSN 1384-5810. doi: 10.1007/s10618-006-0040-z. URL `http://link.springer.com/10.1007/s10618-006-0040-z`. Accessed: 08/19/2018.

[5] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46(3):316–329, jun 1998. ISSN 0030-364X. doi: 10.1287/opre.46.3.316. URL `http://pubsonline.informs.org/doi/abs/10.1287/opre.46.3.316`. Accessed: 08/19/2018.

[6] Sugato. Basu, Ian Davidson, and Kiri Lou. Wagstaff. *Constrained clustering: advances in algorithms, theory, and applications*. CRC Press, 2009. ISBN 9781584889960.

[7] Joachim Behnke, Frank Decker, Florian Grotz, Robert Vehrkamp, Philipp Weinmann, and Verlag Bertelsmann Stiftung. *Reform des Bundestagswahlsystems: Bewertungskriterien und Reformoptionen*. Bertelsmann Stiftung, 2017. ISBN 3867937508.

[8] Mitchell N. Berman. Managing Gerrymandering. *Texas Law Review*, 83, 2004. URL `https://heinonline.org/HOL/Page?handle=hein.journals/tlr83&id=799&div=&collection=`. Accessed: 08/19/2018.

[9] Adrian Bock, Yuri Faenza, Carsten Moldenhauer, and Andres Jacinto Ruiz-Vargas. Solving the Stable Set Problem in Terms of the Odd Cycle Packing Number. In Venkatesh Raman and S P Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 187–198, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

ISBN 978-3-939897-77-4. doi: 10.4230/LIPIcs.FSTTCS.2014.187. URL http://drops.dagstuhl.de/opus/volltexte/2014/4842. Accessed: 08/19/2018.

[10] Andreas Brieden, Peter Gritzmann, and Fabian Klemm. Constrained clustering via diagrams: A unified theory and its application to electoral district design. *European Journal of Operational Research*, 263(1):18–34, nov 2017. ISSN 03772217. doi: 10.1016/j.ejor.2017.04.018. URL http://linkinghub.elsevier.com/retrieve/pii/S037722171730351X. Accessed: 08/19/2018.

[11] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 52–58, 1996. URL https://dl.acm.org/citation.cfm?id=313883. Accessed: 08/19/2018.

[12] Rodolfo Carvajal, Miguel Constantino, Marcos Goycoolea, Juan Pablo Vielma, and Andrés Weintraub. Imposing Connectivity Constraints in Forest Planning Models. *Operations Research*, 61(4), 2013. doi: 10.1287/opre.2013.1183. URL https://doi.org/10.1287/opre.2013.1183. Accessed: 08/19/2018.

[13] Jowei Chen. Unintentional Gerrymandering: Political Geography and Electoral Bias in Legislatures. *Quarterly Journal of Political Science*, 8(3):239–269, jun 2013. ISSN 15540634. doi: 10.1561/100.00012033. URL http://www.nowpublishers.com/article/Details/QJPS-12033. Accessed: 08/19/2018.

[14] R. J. Dakin. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8(3):250–255, mar 1965. ISSN 0010-4620. doi: 10.1093/comjnl/8.3.250. URL https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/8.3.250. Accessed: 08/19/2018.

[15] Dantzig and George B. Origins of the simplex method. *A history of scientific computing*, pages 141–151, 1990. doi: 10.1145/87252.88081. URL https://dl.acm.org/citation.cfm?id=88081. Accessed: 08/19/2018.

[16] George B. Dantzig and Philip Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111, feb 1960. ISSN 0030-364X. doi: 10.1287/opre.8.1.101. URL http://pubsonline.informs.org/doi/abs/10.1287/opre.8.1.101. Accessed: 08/19/2018.

[17] George B. Dantzig and Philip Wolfe. The Decomposition Algorithm for Linear Programs. *Econometrica*, 29(4):767, oct 1961. ISSN 00129682. doi: 10.2307/1911818. URL http://www.jstor.org/stable/1911818?origin=crossref. Accessed: 08/19/2018.

[18] Jacques Desrosiers and Marco E. Lübbecke. A Primer in Column Generation. In *Column Generation*, chapter 1, pages 1–32. Springer-Verlag, New York, 2005. doi: 10.1007/0-387-25486-2_1. URL http://link.springer.com/10.1007/0-387-25486-2_1. Accessed: 08/19/2018.

[19] Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, and François Soumis. Time constrained routing and scheduling. In *Handbooks in Operations Research and*

Management Science, volume 8, chapter 2, pages 35–139. Elsevier, 1995. doi: 10.1016/S0927-0507(05)80106-9. URL `http://linkinghub.elsevier.com/retrieve/pii/S0927050705801069`. Accessed: 08/19/2018.

[20] Derya Dinler and Mustafa Kemal Tural. A Survey of Constrained Clustering. In *Unsupervised Learning Algorithms*, pages 207–235. Springer International Publishing, Cham, 2016. doi: 10.1007/978-3-319-24211-8_9. URL `http://link.springer.com/10.1007/978-3-319-24211-8_9`. Accessed: 08/19/2018.

[21] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, jan 2002. ISSN 0025-5610. doi: 10.1007/s101070100263. URL `http://link.springer.com/10.1007/s101070100263`. Accessed: 08/19/2018.

[22] Adil Fahad, Najlaa Alshatri, Zahir Tari, Abdullah Alamri, Ibrahim Khalil, Albert Y. Zomaya, Sebti Foufou, and Abdelaziz Bouras. A Survey of Clustering Algorithms for Big Data: Taxonomy and Empirical Analysis. *IEEE Transactions on Emerging Topics in Computing*, 2(3):267–279, sep 2014. doi: 10.1109/TETC.2014.2330519. URL `http://ieeexplore.ieee.org/document/6832486/`. Accessed: 08/19/2018.

[23] Julius Farkas. Über die Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik (Crelle's Journal)*, 1902(124):1–27, 1902. ISSN 0075-4102. doi: 10.1515/crll.1902.124.1. URL `https://www.degruyter.com/view/j/crll.1902.issue-124/crll.1902.124.1/crll.1902.124.1.xml`. Accessed: 08/19/2018.

[24] Matteo Fischetti. Facets of two Steiner arborescence polyhedra. *Mathematical Programming*, 51(1-3):401–419, jul 1991. ISSN 0025-5610. doi: 10.1007/BF01586946. URL `http://link.springer.com/10.1007/BF01586946`. Accessed: 08/19/2018.

[25] Matteo Fischetti, Markus Leitner, Ivana Ljubi, Martin Luipersbeck, Michele Monaci, Max Resch, Domenico Salvagnin, and Markus Sinnl. Thinning out Steiner trees : a node-based model for uniform edge costs. *Mathematical Programming Computation*, 9(2):203–229, 2017. ISSN 1867-2949. doi: 10.1007/s12532-016-0111-0.

[26] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, jul 1987. ISSN 00045411. doi: 10.1145/28869.28874. URL `http://portal.acm.org/citation.cfm?doid=28869.28874`. Accessed: 08/19/2018.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[28] Gerald Gamrath. *Generic Branch-Cut-and-Price*. PhD thesis, Technische Universität Berlin, 2010. URL `http://www.zib.de/gamrath/publications/gamrath2010_genericBCP.pdf`. Accessed: 08/19/2018.

[29] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in Constraint Programming. In *Foundations of Artificial Intelligence*, volume 2, chapter 10, pages 329–376. Elsevier, 2006. doi: 10.1016/S1574-6526(06)80014-3. URL `http://linkinghub.elsevier.com/retrieve/pii/S1574652606800143`. Accessed: 08/19/2018.

[30] German Bundestag. Federal Elections Act, 1993. URL `https://bundeswahlleiter.de/en/dam/jcr/4ff317c1-041f-4ba7-bbbf-1e5dc45097b3/bundeswahlgesetz_engl.pdf`. Accessed: 08/19/2018.

[31] Sebastian Goderbauer. *Mathematische Optimierung der Wahlkreiseinteilung für die Deutsche Bundestagswahl*. Springer Fachmedien Wiesbaden, Wiesbaden, 2016. ISBN 978-3-658-15048-8. doi: 10.1007/978-3-658-15049-5. URL `http://link.springer.com/10.1007/978-3-658-15049-5`. Accessed: 08/19/2018.

[32] Sebastian Goderbauer. Political Districting for Elections to the German Bundestag: An Optimization-Based Multi-stage Heuristic Respecting Administrative Boundaries. pages 181–187. 2016. doi: 10.1007/978-3-319-28697-6_26. URL `http://link.springer.com/10.1007/978-3-319-28697-6_26`. Accessed: 08/19/2018.

[33] Sebastian Goderbauer and Marco Lübbecke. A Geovisual Decision Support System for Optimal Political Districting. Working Paper, 2017.

[34] Sebastian Goderbauer and Jeff Winandy. Political Districting Problem: Literature Review and Discussion with regard to Federal Elections in Germany. 2018. In Revision.

[35] Pietro Grilli di Cortona, Cecilia Manzi, Aline Pennisi, Federica Ricca, and Bruno Simeone. *Evaluation and Optimization of Electoral Systems*. Society for Industrial and Applied Mathematics, jan 1999. ISBN 978-0-89871-422-7. doi: 10.1137/1.9780898719819. URL `http://epubs.siam.org/doi/book/10.1137/1.9780898719819`. Accessed: 08/19/2018.

[36] Steve Harenberg, Gonzalo Bello, L. Gjeltema, Stephen Ranshous, Jitendra Harlalka, Ramona Seay, Kanchana Padmanabhan, and Nagiza Samatova. Community detection in large-scale networks: a survey and empirical evaluation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(6):426–439, nov 2014. ISSN 19395108. doi: 10.1002/wics.1319. URL `http://doi.wiley.com/10.1002/wics.1319`. Accessed: 08/19/2018.

[37] Erez Hartuv and Ron Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4-6):175–181, dec 2000. ISSN 0020-0190. doi: 10.1016/S0020-0190(00)00142-3. URL `https://www.sciencedirect.com/science/article/pii/S0020019000001423`. Accessed: 08/19/2018.

[38] Karla Hoffman and Manfred Padberg. Set Covering, Packing and Partitioning Problems. In *Encyclopedia of Optimization*, pages 3482–3486. Springer US, Boston, MA, 2008. doi: 10.1007/978-0-387-74759-0_599. URL `http://www.springerlink.com/index/10.1007/978-0-387-74759-0_599`. Accessed: 08/19/2018.

[39] Ken-ichi Kawarabayashi and Bruce Reed. Odd cycle packing. In *Proceedings of the 42nd ACM symposium on Theory of computing - STOC '10*, page 695, New York, New York, USA, 2010. ACM Press. ISBN 9781450300506. doi: 10.1145/1806689.1806785. URL `http://dl.acm.org/citation.cfm?doid=1806689.1806785`. Accessed: 08/19/2018.

[40] Nan Kong, Andrew J. Schaefer, Brady Hunsaker, and Mark S. Roberts. Maximizing the Efficiency of the U.S. Liver Allocation System Through Region Design. *Management Science*, 56(12):2111–2122, dec 2010. ISSN 0025-1909. doi: 10.1287/mnsc.1100.1249. URL `http://pubsonline.informs.org/doi/abs/10.1287/mnsc.1100.1249`. Accessed: 08/19/2018.

[41] Ivana Ljubić, René Weiskircher, Ulrich Pferschy, Gunnar W. Klau, Petra Mutzel, and Matteo Fischetti. An Algorithmic Framework for the Exact Solution of the Prize-Collecting Steiner Tree Problem. *Mathematical Programming*, 105(2-3):427–449, feb 2006. ISSN 0025-5610. doi: 10.1007/s10107-005-0660-x. URL `http://link.springer.com/10.1007/s10107-005-0660-x`. Accessed: 08/19/2018.

[42] Marco E. Lübbecke. Column generation. *Wiley Encyclopedia of Operations Research and Management Science*, pages 1–19, 2011. ISSN 1550-2376. doi: 10.1002/9780470400531.eorms0158. URL `http://www.or.rwth-aachen.de/research/publications/colgen.pdf`. Accessed: 08/19/2018.

[43] Marco E Lübbecke and Jacques Desrosiers. Selected Topics in Column Generation. *Operations Research*, 53(6):1007–1023, 2005. doi: 10.1287/opre.1050.0234. URL `https://doi.org/10.1287/opre.1050.0234`. Accessed: 08/19/2018.

[44] François Margot. Symmetry in Integer Linear Programming. In *50 Years of Integer Programming 1958-2008*, pages 647–686. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi: 10.1007/978-3-540-68279-0_17. URL `http://link.springer.com/10.1007/978-3-540-68279-0_17`. Accessed: 08/19/2018.

[45] Anuj Mehrotra, Ellis L. Johnson, and George L. Nemhauser. An Optimization Based Heuristic for Political Districting. *Management Science*, 44(8):1100–1114, 1998. ISSN 0025-1909. doi: 10.1287/mnsc.44.8.1100.

[46] Friedrich Pukelsheim. 598 Sitze im Bundestag statt 709? 200 Wahlkreise statt 299! *Deutsches Verwaltungsblatt*, 133(3):153–160, feb 2018. ISSN 2366-0651. doi: 10.1515/dvbl-2018-0306. URL `http://www.degruyter.com/view/j/dvbl.2018.133.issue-3/dvbl-2018-0306/dvbl-2018-0306.xml`. Accessed: 08/19/2018.

[47] Federica Ricca, Andrea Scozzari, and Bruno Simeone. Political districting: from classical models to recent approaches. *4OR*, 9(3):223–254, sep 2011. ISSN 1619-4500. doi: 10.1007/s10288-011-0177-5. URL `http://link.springer.com/10.1007/s10288-011-0177-5`. Accessed: 08/19/2018.

[48] David Ryan and Brian A. Foster. An Integer Programming Approach to Scheduling. *Computer Scheduling of Public Transport*, 1:269, aug 1981.

[49] Satu Elisa Schaeffer and Satu Elisa. Graph clustering. *Computer Science Review*, 1 (1):27–64, aug 2007. doi: 10.1016/j.cosrev.2007.05.001. URL `http://linkinghub.elsevier.com/retrieve/pii/S1574013707000020`. Accessed: 08/19/2018.

[50] A. Schrijver and Alexander. *Theory of linear and integer programming*. Wiley, 1998. ISBN 0471908541. URL `https://dl.acm.org/citation.cfm?id=17634`. Accessed: 08/19/2018.

[51] Nicholas O. Stephanopoulos and Eric M. McGhee. Partisan Gerrymandering and the Efficiency Gap. *University of Chicago Law Review*, 82, 2015. URL `https://heinonline.org/HOL/Page?handle=hein.journals/uclr82&id=843&div=&collection=`. Accessed: 08/19/2018.

[52] Pamela H. Vance, Cynthia Barnhart, Ellis L. Johnson, and George L. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3(2):111–130, may 1994. ISSN 0926-6003. doi: 10.1007/BF01300970. URL `http://link.springer.com/10.1007/BF01300970`. Accessed: 08/19/2018.

[53] François Vanderbeck. *Decomposition and column generation for integer programs*. PhD thesis, Université Catholique de Louvain, 1994.

[54] François Vanderbeck. On Dantzig-Wolfe Decomposition in Integer Programming and ways to Perform Branching in a Branch-and-Price Algorithm. *Operations Research*, 48(1):111–128, feb 2000. ISSN 0030-364X. doi: 10.1287/opre.48.1.111.12453. URL `http://pubsonline.informs.org/doi/abs/10.1287/opre.48.1.111.12453`. Accessed: 08/19/2018.

[55] François Vanderbeck. Branching in branch-and-price: a generic scheme. *Mathematical Programming*, 130(2):249–294, dec 2011. ISSN 0025-5610. doi: 10.1007/s10107-009-0334-1. URL `http://link.springer.com/10.1007/s10107-009-0334-1`. Accessed: 08/19/2018.

[56] François Vanderbeck and Martin W.P. Savelsbergh. A generic view of Dantzig-Wolfe decomposition in mixed integer programming. *Operations Research Letters*, 34(3):296–306, may 2006. ISSN 01676377. doi: 10.1016/j.orl.2005.05.009. URL `http://linkinghub.elsevier.com/retrieve/pii/S0167637705000659`. Accessed: 08/19/2018.

[57] R. R. Vemuganti. Applications of Set Covering, Set Packing and Set Partitioning Models: A Survey. In *Handbook of Combinatorial Optimization*, pages 573–746. Springer US, Boston, MA, 1998. doi: 10.1007/978-1-4613-0303-9_9. URL `http://link.springer.com/10.1007/978-1-4613-0303-9_9`. Accessed: 08/19/2018.

[58] Venice Commission (European Commission for Democracy through Law). Code of Good Practice in Electoral Matters, 2003. URL `http://www.venice.coe.int/webforms/documents/default.aspx?pdffile=CDL-AD(2002)023rev-e`. Accessed: 08/19/2018.

[59] Yiming Wang, Austin Buchanan, and Sergiy Butenko. On imposing connectivity constraints in integer programs. *Mathematical Programming*, pages 1—-31, 2017. ISSN 14364646. doi: 10.1007/s10107-017-1117-8.

[60] Justin C. Williams. Political Districting: A Review. *Papers in Regional Science*, 74(1): 13–40, 1995. ISSN 10568190. doi: 10.1111/j.1435-5597.1995.tb00626.x. URL `http://doi.wiley.com/10.1111/j.1435-5597.1995.tb00626.x`. Accessed: 08/19/2018.

[61] Dongkuan Xu and Yingjie Tian. A Comprehensive Survey of Clustering Algorithms. *Annals of Data Science*, 2(2):165–193, jun 2015. ISSN 2198-5804. doi: 10.1007/s40745-015-0040-1. URL `http://link.springer.com/10.1007/s40745-015-0040-1`. Accessed: 08/19/2018.

Appendices

A. Solving Process Plots

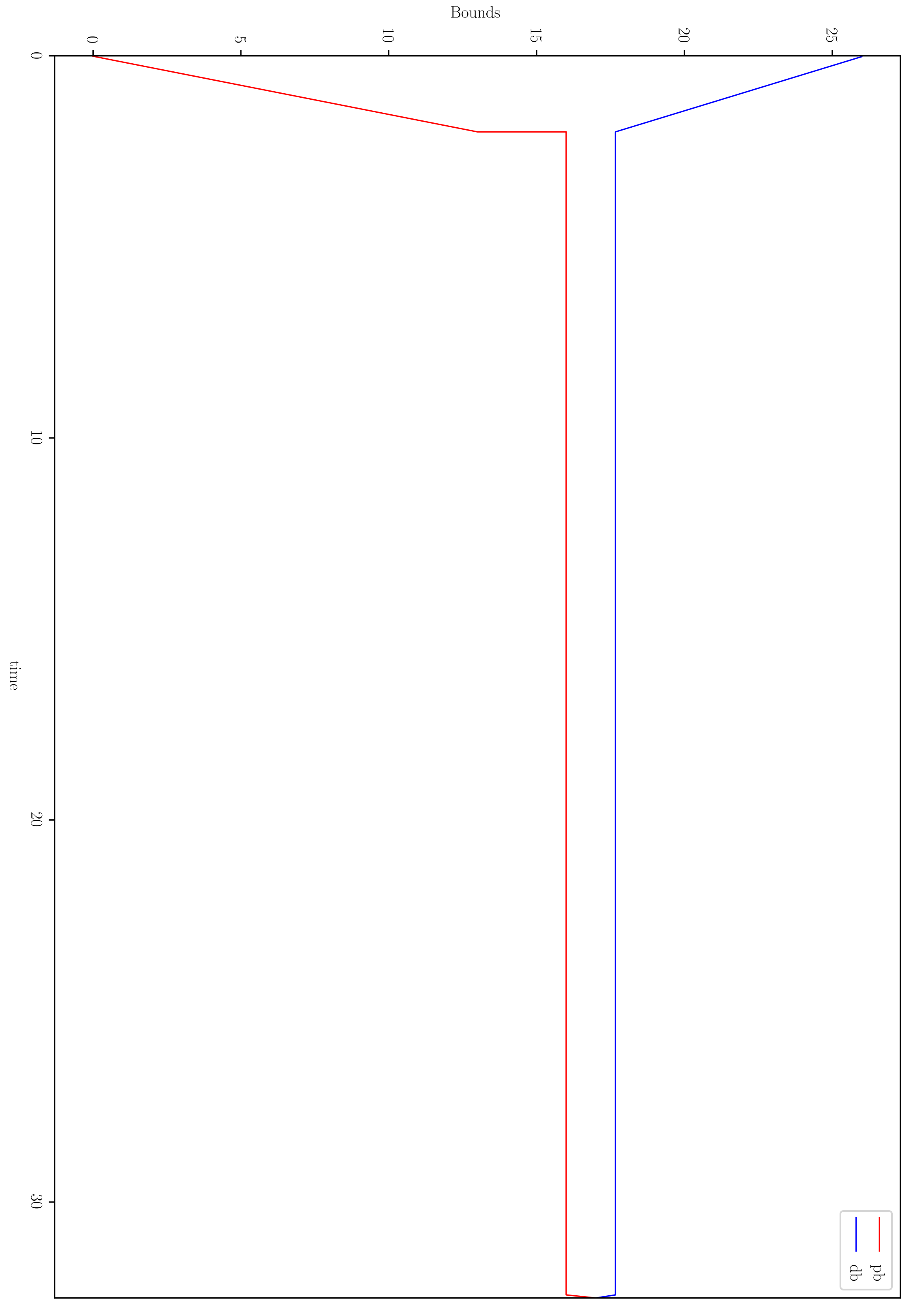


Figure A.1.: *Primal and dual bound over time.*

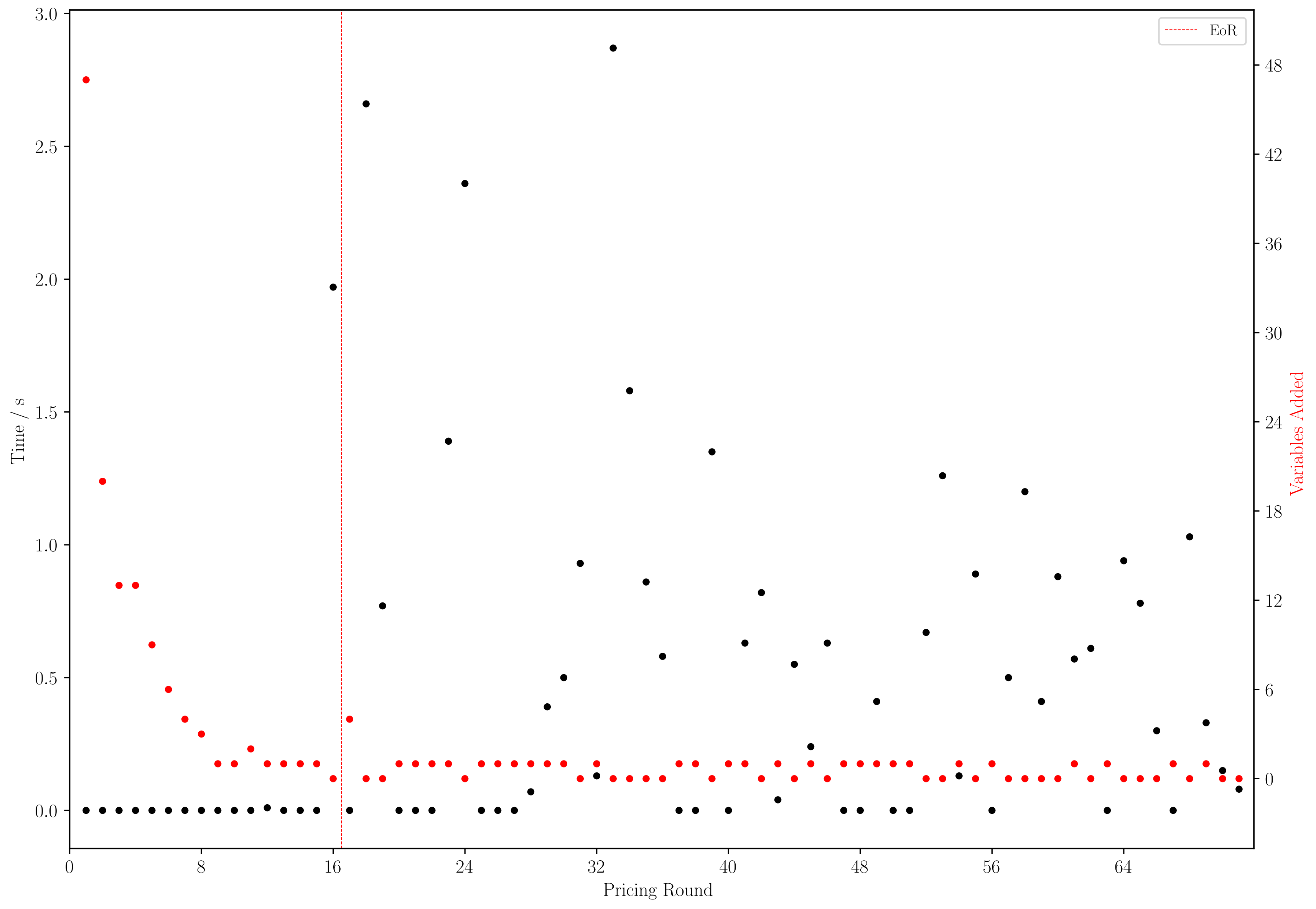


Figure A.2.: *Duration and number of variables added to the RMP for each PR. The vertical red line indicates when the root node LPR was solved to optimality.*

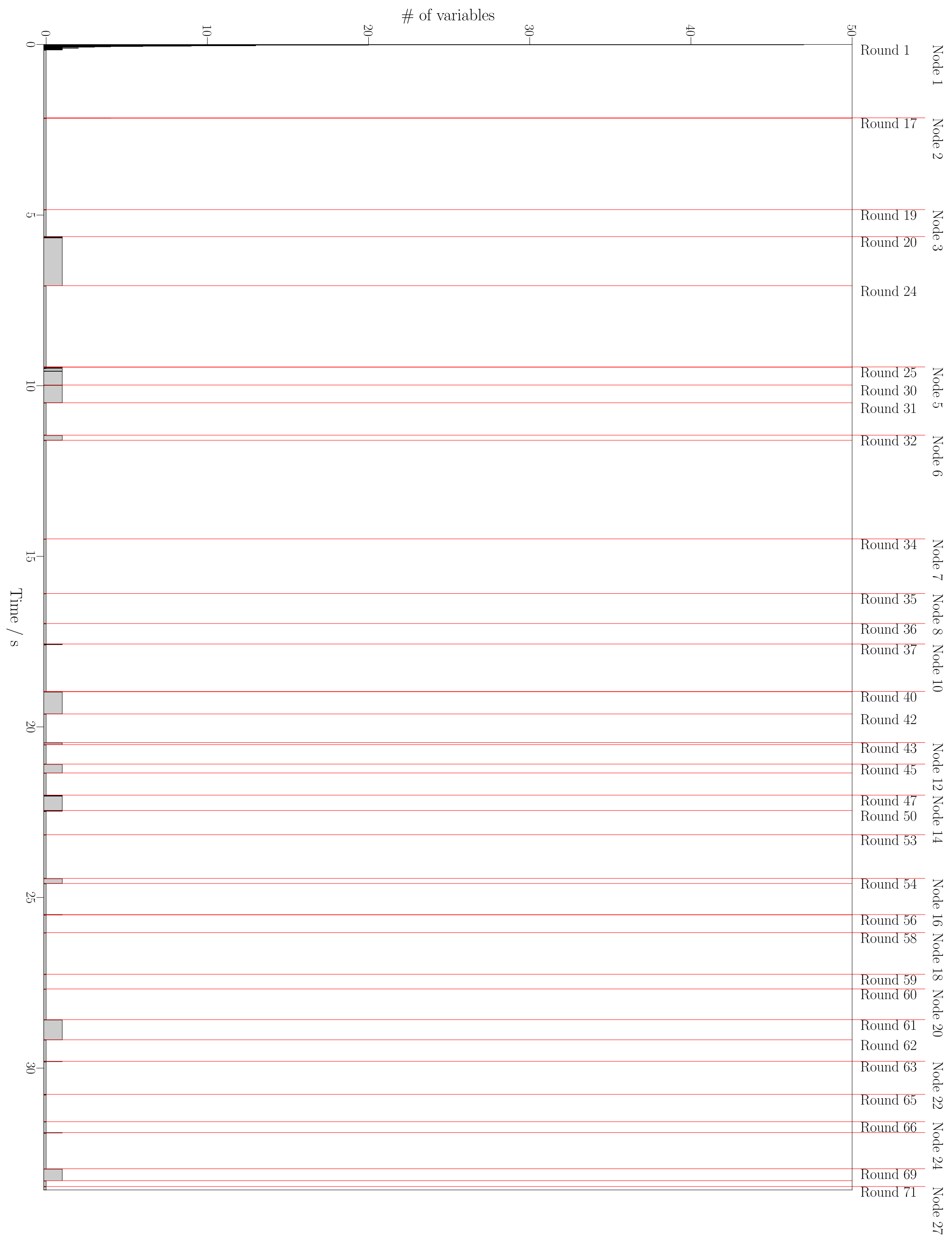


Figure A.3.: *Duration and number of variables added to the RMP for each single PP.*

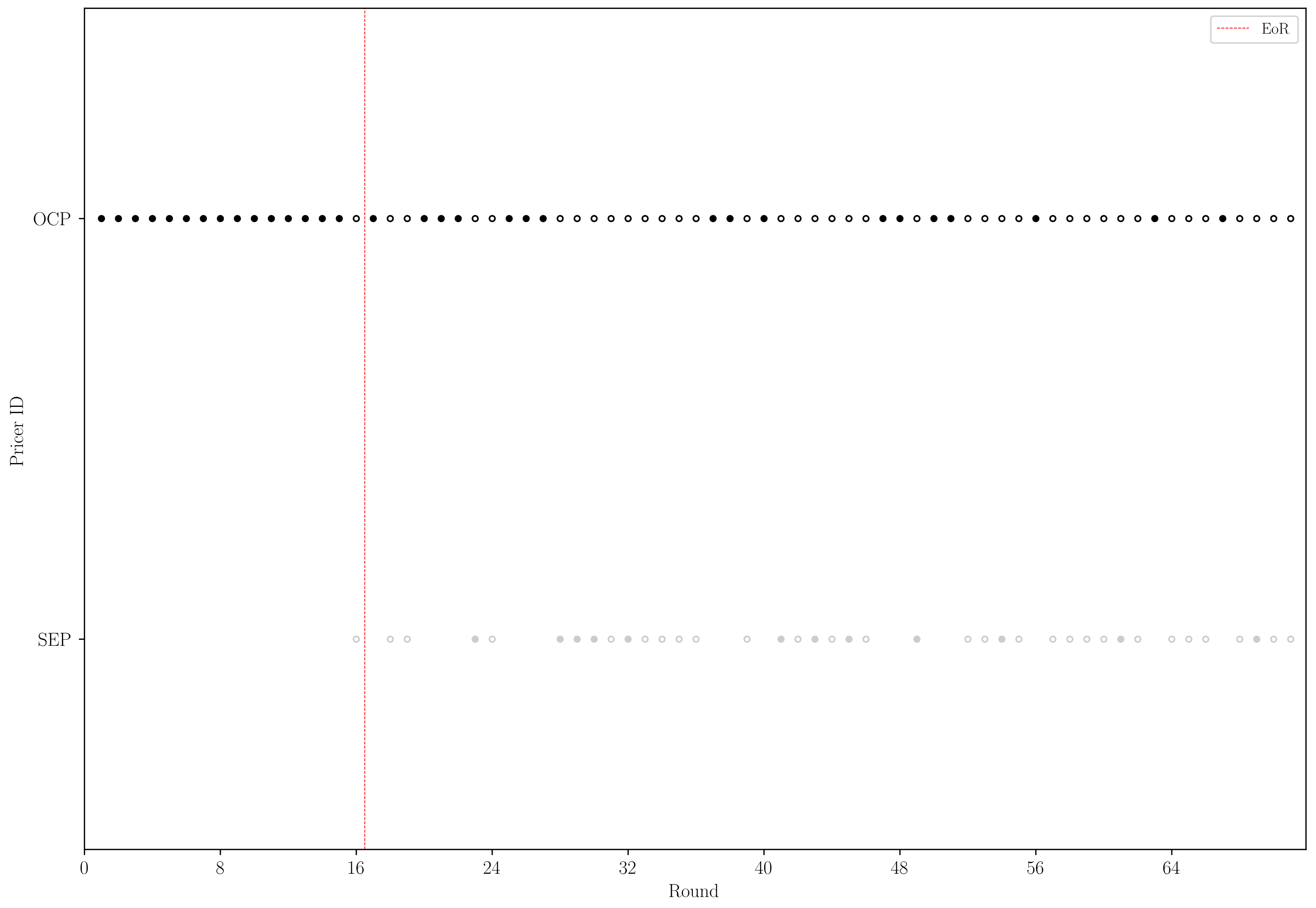


Figure A.4.: *Pricer calls of each pricing round. A filled circle indicates that the pricer succeeded in inserting variables into the RMP, an empty circle that it did not. The vertical red line indicates when the root node LPR was solved to optimality.*

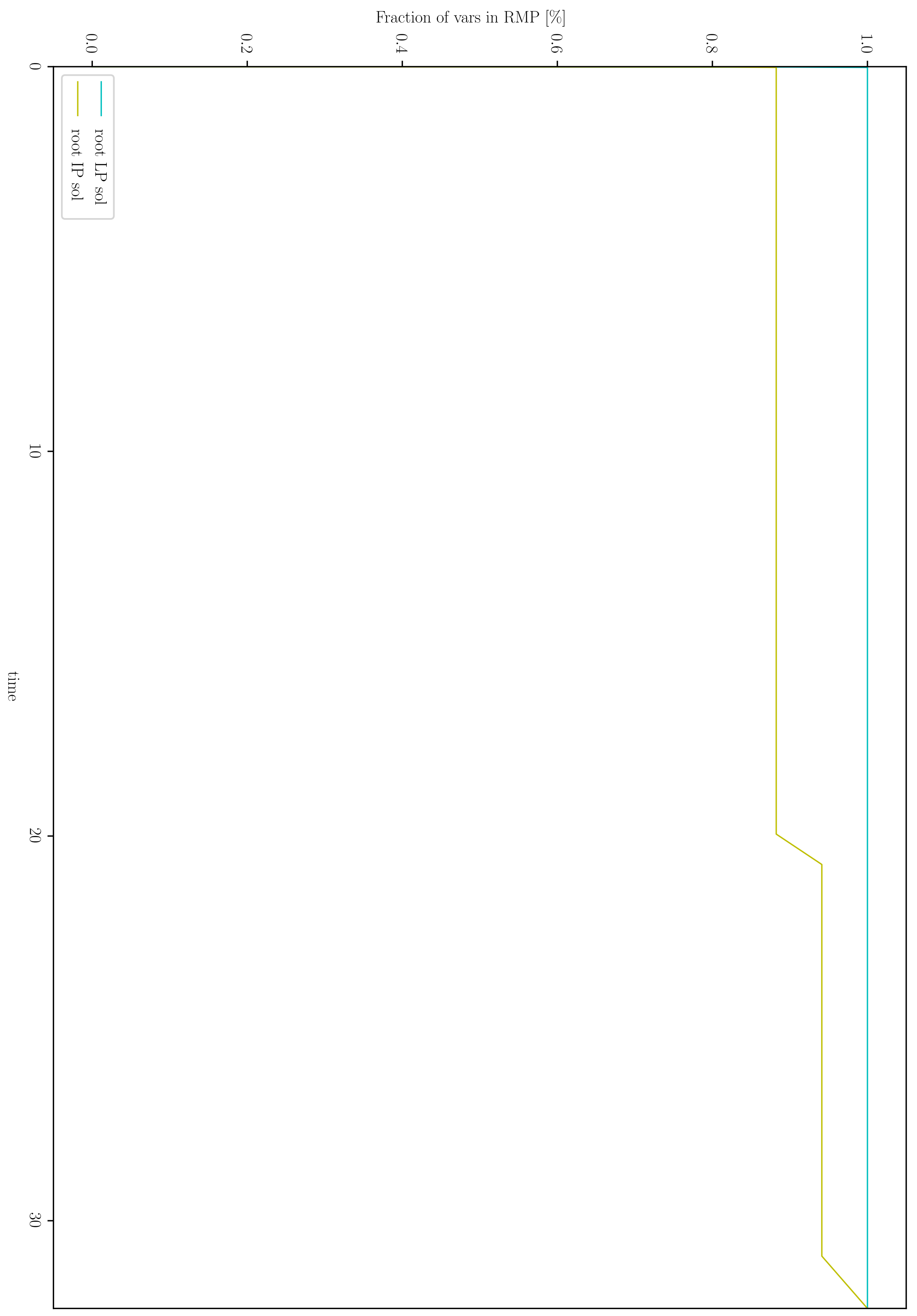


Figure A.5.: *Percentage of variables of the optimal root LP/IP solution that are contained in the RMP.*

B. OCPP Experiments Data

Table B.1.: *OCPP instances.*

State	Nodes	Edges	Components	Nodes in Max. Comp.	Edges in Max. Comp.
anna	138	493	1	138	493
fpsol2.i.1	496	11654	228	269	11654
fpsol2.i.2	451	8691	89	363	8691
fpsol2.i.3	425	8688	63	363	8688
games120	120	638	1	120	638
homer	561	1629	12	542	1620
huck	74	301	3	69	297
inithx.i.1	864	18707	346	519	18707
inithx.i.2	645	13979	88	558	13979
inithx.i.3	621	13969	63	559	13969
jean	80	254	4	77	254
le450_15a	450	8168	1	450	8168
le450_15b	450	8169	1	450	8169
le450_15c	450	16680	1	450	16680
le450_15d	450	16750	1	450	16750
le450_25a	450	8260	1	450	8260
le450_25b	450	8263	1	450	8263
le450_25c	450	17343	1	450	17343
le450_25d	450	17425	1	450	17425
le450_5a	450	5714	1	450	5714
le450_5b	450	5734	1	450	5734
le450_5c	450	9803	1	450	9803
le450_5d	450	9757	1	450	9757
miles1000	128	3216	1	128	3216
miles1500	128	5198	1	128	5198
miles250	128	387	10	92	327
miles500	128	1170	1	128	1170
miles750	128	2113	1	128	2113
mulsol.i.1	197	3925	60	138	3925
mulsol.i.2	188	3885	16	173	3885
mulsol.i.3	184	3916	11	174	3916
mulsol.i.4	185	3946	11	175	3946
mulsol.i.5	186	3973	11	176	3973
myciel3	11	20	1	11	20

myciel4	23	71	1	23	71
myciel5	47	236	1	47	236
myciel6	95	755	1	95	755
myciel7	191	2360	1	191	2360
queen10_10	100	1470	1	100	1470
queen11_11	121	1980	1	121	1980
queen12_12	144	2596	1	144	2596
queen13_13	169	3328	1	169	3328
queen14_14	196	4186	1	196	4186
queen15_15	225	5180	1	225	5180
queen16_16	256	6320	1	256	6320
queen5_5	25	160	1	25	160
queen6_6	36	290	1	36	290
queen7_7	49	476	1	49	476
queen8_12	96	1368	1	96	1368
queen8_8	64	728	1	64	728
queen9_9	81	1056	1	81	1056
school1	385	19095	5	377	19091
school1_nsh	352	14612	5	344	14608
zeroin.i.1	211	4100	86	126	4100
zeroin.i.2	211	3541	55	157	3541
zeroin.i.3	206	3540	50	157	3540

Table B.2.: *OCPP solving time. Missing values indicate that the time limit was reached.*

State	$t_{SIP,p}$	$t_{SPSP,p}$	$t_{SIP+TCI,p}$	$t_{SIP+TCP,p}$
anna	5.05	55.18	0.36	0.24
fpsol2.i.1				
fpsol2.i.2			2,751.95	
fpsol2.i.3				
games120				
homer				
huck	21.17	44.46	19.15	62.58
inithx.i.1				
inithx.i.2				
inithx.i.3				
jean	53.71	64.66	14.87	32.51
le450_15a				
le450_15b				
le450_15c				
le450_15d				
le450_25a				
le450_25b				
le450_25c				
le450_25d				
le450_5a				
le450_5b				
le450_5c				
le450_5d				
miles1000	776.76		40.73	0.43
miles1500	1,118.56		8.74	18.13
miles250	125.52	1,972.45	110.84	23.19
miles500			1,945.49	102.33
miles750	267.18		12.34	5.43
mulsol.i.1	4,796.57		198.93	241.42
mulsol.i.2				
mulsol.i.3	2,846.47		931.85	2,402.16
mulsol.i.4				
mulsol.i.5	1,658.86			1,451.35
myciel3	1.09	1.12	1.10	1.11
myciel4	3.28	11.27	3.29	3.25
myciel5	227.62	178.59	227.24	228.23
myciel6	4,905.58		4,902.04	4,905.38
myciel7				
queen10_10	177.62		0.18	1.16
queen11_11	292.14		0.30	7.48
queen12_12	4,468.12		0.47	2,436.24
queen13_13	1,531.72		0.50	8.46
queen14_14	1,521.50		0.65	10.94

queen15_15			0.83	
queen16_16	2,499.03		1.15	200.88
queen5_5	0.77	2.80	0.05	0.02
queen6_6	28.94	128.34	0.08	4.22
queen7_7	26.67	240.62	0.04	0.07
queen8_12	668.21		0.17	785.64
queen8_8	41.79	1,342.47	0.21	0.07
queen9_9	492.97		0.13	374.17
school1				
school1_nsh				
zeroin.i.1			1,169.69	132.82
zeroin.i.2	2,139.37		1,084.63	4,741.41
zeroin.i.3	822.50		544.41	1,235.50

Table B.3.: *OCPP gap.*

State	$g_{SIP,p}$	$g_{SPSP,p}$	$g_{SIP+TCI,p}$	$g_{SIP+TCP,p}$
anna	0.0000	0.0000	0.0000	0.0000
fpsol2.i.1	31.5889	54.0000	0.0307	0.1070
fpsol2.i.2	0.0524	29.0000	0.0000	0.0407
fpsol2.i.3	0.0278	22.5000	0.0165	0.0165
games120	0.0526	0.0526	0.0256	0.0526
homer	0.0684	0.0965	0.0549	0.0396
huck	0.0000	0.0000	0.0000	0.0000
inithx.i.1	33.9013	71.0000	0.0321	0.0625
inithx.i.2	0.0794	25.8750	0.0226	0.0382
inithx.i.3	0.0560	50.7500	0.0144	0.0225
jean	0.0000	0.0000	0.0000	0.0000
le450_15a	0.0700	5.5217	0.1103	0.0700
le450_15b	0.1344	5.8182	0.1003	0.0920
le450_15c	1.0000	2.4091	0.0490	0.0714
le450_15d	4.5556	3.5455	0.0490	0.0714
le450_25a	0.1583	20.4286	0.0296	0.0692
le450_25b	0.1452	7.8235	0.1270	0.0597
le450_25c	2.6585	24.0000	0.0417	0.0638
le450_25d	5.2500	6.1429	0.0345	0.0791
le450_5a	0.0949	9.7143	0.0135	0.0274
le450_5b	0.1278	4.1724	0.0204	0.2195
le450_5c	0.1538	4.1724	0.0714	0.0791
le450_5d	0.1364	5.0000	0.0638	0.0714
miles1000	0.0000	1.3333	0.0000	0.0000
miles1500	0.0000	2.2308	0.0000	0.0000
miles250	0.0000	0.0000	0.0000	0.0000
miles500	0.0244	0.1351	0.0000	0.0000
miles750	0.0000	0.3548	0.0000	0.0000
mulsol.i.1	0.0000	0.9118	0.0000	0.0000
mulsol.i.2	0.0362	5.2000	0.0362	0.0362
mulsol.i.3	0.0000	2.3889	0.0000	0.0000
mulsol.i.4	0.0284	9.1667	0.0284	0.0248
mulsol.i.5	0.0000	1.2963	0.0399	0.0000
myciel3	0.0000	0.0000	0.0000	0.0000
myciel4	0.0000	0.0000	0.0000	0.0000
myciel5	0.0000	0.0000	0.0000	0.0000
myciel6	0.0000	0.6000	0.0000	0.0000
myciel7	0.3354	20.0000	0.3354	0.3354
queen10_10	0.0000	0.0645	0.0000	0.0000
queen11_11	0.0000	0.0811	0.0000	0.0000
queen12_12	0.0000	0.3333	0.0000	0.0000
queen13_13	0.0000	0.1429	0.0000	0.0000
queen14_14	0.0000	0.3265	0.0000	0.0000

queen15_15	0.1719	0.3889	0.0000	0.0563
queen16_16	0.0000	0.1486	0.0000	0.0000
queen5_5	0.0000	0.0000	0.0000	0.0000
queen6_6	0.0000	0.0000	0.0000	0.0000
queen7_7	0.0000	0.0000	0.0000	0.0000
queen8_12	0.0000	0.0667	0.0000	0.0000
queen8_8	0.0000	0.0000	0.0000	0.0000
queen9_9	0.0000	0.0385	0.0000	0.0000
school1	20.3333	31.0000	0.0246	0.0331
school1_nsh	0.3256	15.7143	0.0179	0.0654
zeroin.i.1	0.0375	1.1875	0.0000	0.0000
zeroin.i.2	0.0000	4.8333	0.0000	0.0000
zeroin.i.3	0.0000	2.5789	0.0000	0.0000

Table B.4.: *OCPP PR time and unsuccessful PR time. Missing values indicate that all pricing rounds were successful.*

State	$t^{PR}_{SIP,p}$	$t^{PR}_{SPSP,p}$	$t^{UPR}_{SIP,p}$	$t^{UPR}_{SPSP,p}$
anna	0.1141	0.8110	0.0300	24.7100
fpsol2.i.1	22.3587	1,827.6250		2,194.8900
fpsol2.i.2	21.6852	603.8458	143.4088	129.9900
fpsol2.i.3	20.5115	81.5763	154.3185	597.6200
games120	5.5707	13.7663	6.4330	41.2599
homer	8.1059	23.8886	11.3536	801.6650
huck	0.2859	0.5045	0.7313	4.1343
inithx.i.1	16.9004	27.7407		176.2700
inithx.i.2	16.9000	19.7425	229.1221	611.9900
inithx.i.3	16.4748	26.6758	103.6223	107.7700
jean	0.2563	0.6592	0.3154	2.8259
le450_15a	14.9619	172.2626	199.2572	36.7700
le450_15b	15.5774	164.5039	290.2400	126.0700
le450_15c	18.6026	63.5590	16.1000	518.0600
le450_15d	19.0963	102.2931	6.8400	343.2900
le450_25a	16.8561	249.7217	145.0913	129.3000
le450_25b	16.3951	249.4838	137.8713	56.4600
le450_25c	18.8462	292.0224	13.5000	390.0200
le450_25d	18.4588	244.3017		607.0100
le450_5a	9.5055	136.2983	19.9866	29.3800
le450_5b	11.9522	76.8552	141.3172	1,998.9400
le450_5c	14.8407	134.1219	330.7600	136.8400
le450_5d	14.3375	127.0181	325.1325	90.4800
miles1000	3.6106	59.5444	0.0752	730.1200
miles1500	6.8174	120.2312	0.1032	188.4200
miles250	0.7507	4.0490	1.4526	7.8066
miles500	0.7063	43.3775	0.7042	402.5063
miles750	1.3473	59.0196	0.0614	2,337.7350
mulsol.i.1	20.5841	47.6899	77.5788	2,563.0300
mulsol.i.2	15.8922	45.9111	25.1268	56.2600
mulsol.i.3	11.2938	60.5669	24.6427	679.0600
mulsol.i.4	9.9698	72.8167	13.1060	164.8500
mulsol.i.5	8.1308	75.8983	24.9193	41.7700
myciel3	0.0540	0.0379	0.0800	0.0864
myciel4	0.1924	0.2813	1.2000	1.2250
myciel5	2.3953	2.5149	4.2636	15.5020
myciel6	22.8131	67.9280	39.4966	391.6119
myciel7	39.9955	54.1465	372.1973	4,256.5800
queen10_10	0.9736	50.7044	0.0351	475.7221
queen11_11	1.1843	49.6692	0.3257	713.9111
queen12_12	8.5378	39.5735	15.2014	1,186.5980
queen13_13	2.6210	38.7124	0.0721	1,494.6950

queen14_14	2.8910	30.5271	0.0797	1,656.3667
queen15_15	27.4780	27.7058	261.9013	2,168.0600
queen16_16	3.4991	23.6346	0.1220	1,729.9750
queen5_5	0.0317	0.1463	0.0050	
queen6_6	0.4584	1.8868	0.7981	9.1033
queen7_7	0.3059	3.8802	0.0150	32.6083
queen8_12	2.5869	53.7367	3.6061	445.0800
queen8_8	0.4393	13.1605	0.0206	93.7323
queen9_9	2.7680	52.1733	6.2632	252.6941
school1	20.3373	1,240.8167		1,068.6300
school1_nsh	21.2987	615.5492	1,118.1500	749.2400
zeroin.i.1	27.6903	75.8875	75.0038	73.0300
zeroin.i.2	10.5897	101.5155	31.1250	15.1200
zeroin.i.3	6.0027	97.4039	34.0233	44.5900

C. GPDP Experiments Data

Table C.1.: *GPDP instances.*

State	Nodes	Edges
Baden-Wuerttemberg	612	1727
Bayern	1536	4494
Berlin	62	159
Brandenburg	200	511
Bremen	24	52
Hamburg	100	249
Hessen	474	1290
Mecklenburg-Vorpommern	118	273
Niedersachsen	490	1321
Nordrhein-Westfalen	689	1935
Rheinland-Pfalz	209	557
Saarland	52	128
Sachsen	439	1216
Sachsen-Anhalt	124	322
Schleswig-Holstein	171	417
Thueringen	219	574

Table C.2.: *GPDP solving time. Missing values indicate that the time limit was reached.*

State	$t_{SIP,p}$	$t_{SPSP,p}$	$t_{SIP+GI,p}$	$t_{SIP+GP,p}$
Baden-Wuerttemberg				
Bayern				
Berlin	1,019.30	331.57	20.29	239.30
Brandenburg				
Bremen	37.16	89.32	3.27	4.29
Hamburg				
Hessen				
Mecklenburg-Vorpommern				
Niedersachsen				
Nordrhein-Westfalen				
Rheinland-Pfalz				
Saarland	5,012.78	1,415.49	19.85	5,381.89
Sachsen				
Sachsen-Anhalt				
Schleswig-Holstein				
Thueringen				

Table C.3.: *GPDP gap. Missing values indicate infinite gap.*

State	$g_{SIP,p}$	$g_{SPSP,p}$	$g_{SIP+GI,p}$	$g_{SIP+GP,p}$
Baden-Wuerttemberg			0.0551	0.0552
Bayern				0.0719
Berlin	0.0000	0.0000	0.0000	0.0000
Brandenburg			0.1380	
Bremen	0.0000	0.0000	0.0000	0.0000
Hamburg		0.0005	0.0181	0.0181
Hessen			0.0704	0.0704
Mecklenburg-Vorpommern			0.1255	0.1133
Niedersachsen				0.0510
Nordrhein-Westfalen				0.0716
Rheinland-Pfalz			0.0635	0.0635
Saarland	0.0000	0.0000	0.0000	0.0000
Sachsen			0.0442	0.0419
Sachsen-Anhalt			0.0711	0.0711
Schleswig-Holstein			0.0956	0.0903
Thueringen			0.0296	0.0296

Table C.4.: *GPDP PR time and unsuccessful PR time. Missing values indicate that all pricing rounds were successful.*

State	$t^{PR}_{SIP,p}$	$t^{PR}_{SPSP,p}$	$t^{UPR}_{SIP,p}$	$t^{UPR}_{SPSP,p}$
Baden-Wuerttemberg	337.5894	25.7407		71.1100
Bayern	568.4353	75.0100	4,227.8800	402.6900
Berlin	2.8381	0.7373	25.5400	43.5800
Brandenburg	635.2965	3.0364		
Bremen	0.1711	0.2892	1.5000	28.1100
Hamburg	17.6022	2.2859	10.3700	173.0200
Hessen	720.1893	11.6310	10,257.5800	31.7200
Mecklenburg-Vorpommern	24.5140	2.3077		
Niedersachsen	490.9582	12.5182		36.4500
Nordrhein-Westfalen	220.4420	19.1132		154.5200
Rheinland-Pfalz	514.3029	4.5900		9.5200
Saarland	3.5999	1.1473	244.9900	87.8000
Sachsen	568.5095	9.5927	3,936.6600	21.9800
Sachsen-Anhalt	50.2157	2.5923	190.3100	
Schleswig-Holstein	83.7137	3.1420	449.9100	
Thueringen	136.7019	4.6767	2.6000	