Template LaTeX Wiki von BAzubis für BAzubis

Projektarbeit 1 (T3_2000)

im Rahmen der Prüfung zum

Master of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Vorname Nachname

Abgabedatum:	01. Februar 2025
Bearbeitungszeitraum:	01.10.2024 - 31.01.2025
Matrikelnummer, Kurs:	0000000, TINF15B1
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Betreuer der Ausbildungsfirma:	B-Vorname B-Nachname
Gutachter der Dualen Hochschule:	DH-Vorname DH-Nachname

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit 1 (T3_2000) mit dem Thema:

Template LaTeX Wiki von BAzubis für BAzubis

gemäß § 5 der "Studien- und Prüfungsordnung DHBW Technik" vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den August 22, 2025

Nachname, Vorname

Abstract

- English -

This is the starting point of the Abstract. For the final bachelor thesis, there must be an abstract included in your document. So, start now writing it in German and English. The abstract is a short summary with around 200 to 250 words.

Try to include in this abstract the main question of your work, the methods you used or the main results of your work.

Abstract

- Deutsch -

Dies ist der Beginn des Abstracts. Für die finale Bachelorarbeit musst du ein Abstract in deinem Dokument mit einbauen. So, schreibe es am besten jetzt in Deutsch und Englisch. Das Abstract ist eine kurze Zusammenfassung mit ca. 200 bis 250 Wörtern.

Versuche in das Abstract folgende Punkte aufzunehmen: Fragestellung der Arbeit, methodische Vorgehensweise oder die Hauptergebnisse deiner Arbeit.

Contents

Formelverzeichnis

A	mm^2	Fläche
D	mm	Werkstückdurchmesser
$d_{\mathbf{min}}$	mm	kleinster Schaftdurchmesser
$L_{\mathbf{1}}$	mm	Länge des Werkstückes Nr. 1
	Grad	Freiwinkel
	Grad	Keilwinkel

Abkürzungsverzeichnis

GCG	Generic Column Generation
SCIP	SCIP
BaPCod	Branch-and-price Code
hMETIS	Hypergraph METIS
strIPlib	Structured Integer Program Library
MIP	Mixed Integer Programming

List of Figures

List of Tables

Quellcodeverzeichnis

1 Introduction

1.1 Motivation and Contribution

1.2 Structure

2 Preliminaries

In this chapter, we provide sufficient background needed to understand the algorithmic details and ideas discussed in Chapter 5. First, we take a look at linear optimization as a general concept and discuss necessary basics such as Mixed Integer Programming (MIP). Afterwards, these concepts are used to take a more detailed look into the characteristics of the Dantzig-Wolfe decomposition, which represents a major component of the decomposition solver GCG. In addition, notations and basic definitions related to Graph Theory are introduced including the concept of *Partition Refinement*; an algorithmic approach used as a building block in Chapters 5 and 6.

2.1 Linear Optimization

Linear Optimization is a mathematical optimization technique used to determine the best possible values for a set of variables in a given model, whose constraints or requirements are represented by linear relationships. The goal is typically to maximize or minimize a objective function, subject to a set of equality and/or inequality constraints. Both objective and constraints must be linear.

If all variables are only allowed to take values from $\mathbb{R}^n_{\geq}$, i.e., only continuous values, then this optimization technique is referred to as *Linear Programming*. In a standard form, a linear programming problem with variable vector $\mathbf{x} \in \mathbb{R}^n$, constraint matrix $A \in \mathbb{R}^{m \times n}$, objective coefficients $c \in \mathbb{R}^n$ and right-hand side vector of the constraints $b \in \mathbb{R}^m$ can be expressed as follows:

$$z^*_{LP} = \min \quad c^T \mathbf{x} \tag{2.1}$$

$$\text{s.t.} \quad A\,\mathbf{x} \geq b \tag{2.2}$$

$$\mathbf{x} \geq 0 \tag{2.3}$$

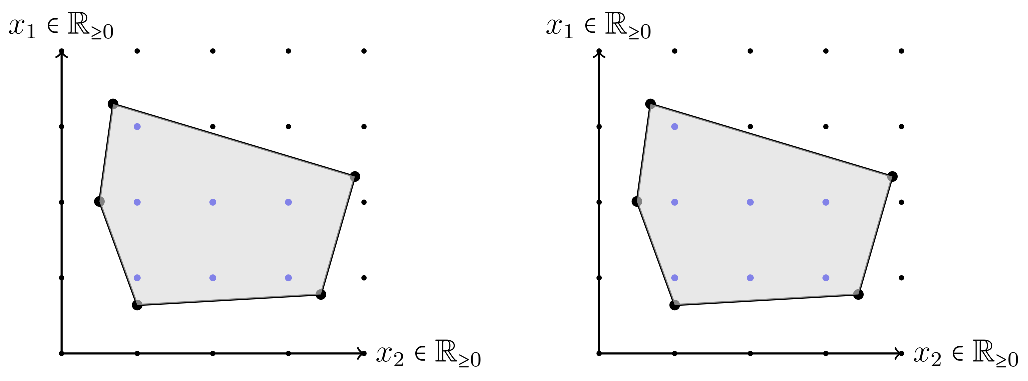

$x_1 \in \mathbb{R}_{\geq 0}$
$x_2 \in \mathbb{R}_{\geq 0}$
$x_1 \in \mathbb{R}_{\geq 0}$
$x_2 \in \mathbb{R}_{\geq 0}$

Figure 2.1: Solution space of a linear program highlighted in gray. Point highlighted in blue represent the solution space of the corresponding integer linear program.

unbounded on right

Note that it is also possible to represent a set of equality $A'\mathbf{x} = b'$ by the two sets of inequalities $A'\mathbf{x} \geq b'$ and $A'\mathbf{x} \leq b'$. Without loss of generality, we assume optimization problems to always be minimization problems, unless explicitly stated otherwise. Constraints 2.2 specify a *convex* polytope over which the Objective Function 2.1 is optimized as shown in Figure 2.1. A solution vector $\mathbf{x} \in \mathbb{R}_{\geq 0}$ is called *feasible*, iff it satisfies both both constraints 2.2 and 2.3. The linear program as a whole is called *feasible*, if there exists such a vector, otherwise it is considered *infeasible*. A feasible solution $\mathbf{x} \in \mathbb{R}_{\geq 0}$ is called *optimal* iff

$$c^T\mathbf{x} = \min\{c^T\mathbf{x} \mid \mathbf{x} \text{ is feasible}\} \iff \nexists\, \mathbf{y} \in \mathbb{R}_{\geq 0} : (\mathbf{y} \text{ feasible}) \wedge \underbrace{\left(c^T\mathbf{y} < c^T\mathbf{x}\right)}_{\mathbf{y} \text{ is ``better''}}$$

If there exists a feasible solution vector, but no optimal one, then the problem is called *unbounded*, as shown in Figure 2.1.

Linear programming is widely used in various fields such as operations research, economics, engineering, and logistics, due to its efficiency in solving large-scale real-world optimization problems. Algorithms such as the Simplex Method and Interior Point Methods are commonly used to solve LP problems efficiently. The simplex algorithm in particular is widely used in practice because of its efficiency on most problems, even though its worst-case complexity is exponential-time on certain families of problems depending on the chosen pivot-rule. However, on most problems the simplex algorithm only takes a polynomial number of steps to terminate. For more information about how the algorithms work and their mathematical details we refer to [1].

cite

2.1.1 Mixed-Integer Programs

$$\begin{aligned} z^*_{LP} = \min \quad & c^T\mathbf{x} && + && d^T\mathbf{y} \\ \text{s.t.} \quad & A\,\mathbf{x} && + && B\,\mathbf{y} \geq b \\ & \mathbf{x} && && \in \mathbb{R}_{\geq 0} \\ & && && \mathbf{y} \in \mathbb{Z}_{\geq 0} \end{aligned}$$

Even thought linear programs are often sufficient to model a certain problem

2.2 Dantzig-Wolfe Decomposition

The *Dantzig-Wolfe Decomposition* is a

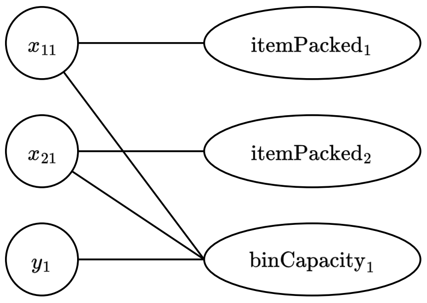
2.3 Graph Theory

Graphs are the fundamental data structure used in almost every aspect of computer science. This section will *not* introduce new concepts not already found in standard literature about graphs and related topics. We will mainly introduce the notation used for the following sections and chapters.

A *graph* is a tuple $G = (V, E)$ with $V \subseteq \{1, 2, \dots, n\}$ for some $n \in \mathbb{N}$ and $E \subseteq \{(u, v) \mid (u, v) \in V \times V\}$. The elements of set V are called *vertices* or *nodes*. The elements of set are ordered pairs called *directed edges*, *arcs* or simply *edges* which connect two vertices with each other. The set of outgoing neighbors of a specific vertex $v \in V$ is denoted $E(v) = \{(v, v') \mid v' \in V, vEv'\}$. The set of incoming neighbors $E^{-1}(v) = \{(v', v) \mid v' \in V, v'Ev\}$ is defined analogously. In this thesis, we will distinguish four types of graphs: directed, un-directed and bipartite, with directed being the assumed type if not stated explicitly.

For *un-directed* graphs, the edge relation E must be symmetric $\forall u, v \in V : uEv \rightarrow vEu$, i.e., if vertex u is connected to v or vice versa, then the corresponding back-edge must exists as well.

Bi-partite graphs are a special kind of graph class, where the set V can be represented with two sets $L, R \subseteq V$ such that $L \cap R = \varnothing$ and $E \subseteq \{(u, v) \mid u \in L, v \in R\}$. More informally, the vertex V can be split into two disjoint subsets $L, R \subseteq V$ such that no edges exists between vertices in each corresponding set. Bi-partite graphs are especially interesting, because the relationship between variables and the constraints they participate in can be encoded as such a graph as shown in Figure 2.2. This concept will be used in Chapter 4 to algorithmically detect various underlying structures. Wording



$$
\begin{array}{llr l}
\text{min} & & \sum_{j=1}^{m} y_j & \\
\text{s.t.} & & x_{11} = 1 & \text{itemPacked}_1 \\
& & x_{21} = 1 & \text{itemPacked}_2 \\
& & 100x_{11} + 99x_{21} \leq 200y_1 & \text{binCapacity}_1 \\
& & x_{11}, x_{21} \in 0, 1 & \\
& & y_1 \in 0, 1 &
\end{array}
$$

Figure 2.2: A simple binpacking problem with 2 items and 1 bin represented as graph.

2.4 Partition Refinement

Partition refinement is a fundamental concept in computer science, particularly relevant in fields such as automata theory [2], graph theory, and model checking [3]. A *partition* refers to a decomposition of a finite set U into disjoint, non-empty subsets $\{A_1, A_2, \ldots, A_k\}$, called *cells* or *blocks*, such that:

$$\bigcup_{i=0}^{k} A_i = U \text{ and } \forall i \neq j : A_i \cap A_j = \varnothing$$

The set of all partitions over a set U is denoted $\Pi(U)$. A partition $\pi = \{A_1, A_2, \ldots, A_k\}$ of a set U is called a refinement of another partition $\pi' = \{B_1, B_2, \ldots, B_m\}$, denoted $\pi \sqsubseteq \pi'$, iff

$$\forall A_i \in \pi \;\; \exists B_j \in \pi' : A_i \subseteq B_j$$

As a special case, a partition is a refinement of itself. More informally, partition π' must reflect a "finer" classification of the elements than in π.

Partition refinement refers to an *iterative* process that refines a given initial partition of a set over the course of multiple iterations. In the following, let $f : P \times Q \mapsto \Pi(A)$ be a function, which partitions the elements from $P \subseteq U$ with respect to the elements in $Q \subseteq U$. The arguments P and Q are called *target cell* and *inducing cell* respectively. A partition π is called *stable* with respect to f, iff

$$\forall A_i, A_j \in \pi : \; |f(A_i, A_j)| = 1$$

That is, there is no cell in π which acts as a "splitter" to another cell according to f. Let π_{init} be an *initial* partition. The goal is typically to find the coarsest partition $\pi_f = \{A_1, A_2, \ldots, A_k\}$ of U such that the following properties hold:

1. The partition π_f is a *refinement* of the initial partition π_{init}
2. The partition π_f is *stable* with respect to f.

In the following, we define *Step* : $\Pi(U) \mapsto \Pi(U)$ as function performing one refinement step, i.e., it picks a splitter-cell B_j if it exists and replaces each cell B_i of the input partition with $f(B_i, B_j)$.

Algorithm 1 A simple partition refinement algorithm which refines π_{init} until a fixed-point is reached.

Input: Initial partition $\Pi_{init} = \{A_1, A_2, \dots, A_k\}$, monotone splitter-function $f : P \times Q \mapsto \Pi(P)$
Output: Coarsest stable partition

```
function IterateRefinement($\Pi_{init}, f$)
    $i \leftarrow 0$
    $\Pi_0 \leftarrow \Pi_{init}$
    repeat
        $i \leftarrow i + 1$
        $\Pi_i \leftarrow Step(\Pi_{i-1})$
    until $\Pi_i = \Pi_{i-1}$
    return $\Pi_i$
end function
```

This process is illustrated in Algorithm 1. Note that new cells are continuously being produced in the loop which are able to act as inducing cells during the next iteration.

For the purposes of this work, f will usually represent a function structurally similar to a *connection function* as it used in many graph automorphism packages. Given a graph $G = (V, E)$, then we define two types of connection function as follows:

$$f_{\text{count}}(v, X_{\text{ind}}) = |\{v' \in V \mid \forall (v, v') \in E, v' \in X_{\text{ind}}\}| \tag{2.4}$$

$$f_{\text{exists}}(v, X_{\text{ind}}) = \begin{cases} 1 & f_{\text{count}}(v, X_{\text{ind}}) \geq 1 \\ 0 & \text{else} \end{cases} \tag{2.5}$$

If Function 2.5 is used, then the problem of finding the coarsest partition with respect to f is equivalent to the *Relational coarsest partition problem* described in [4] which also contains corresponding correctness and termination proofs. For this case in particular, Algorithm 1 always maintains the invariant $\Pi_i \sqsubseteq \Pi_{i-1}$.

Furthermore, the underlying problem structure to which partition refinement is applied, as well as the type of splitter function used, are not inherently restricted. In practice, however, many problems can be reformulated or encoded as graphs, where the function f captures a vertex property. For instance, in deterministic finite automaton (DFA) minimization, partition refinement is used to iteratively distinguish states by observing the equivalence classes of their transitions (Hopcroft's algorithm): two states are grouped together only if, for every input symbol, their transitions lead into the same partition class; in graph isomorphism testing, it could encode vertex degrees or local neighborhood structures; and in Markov decision processes (MDPs), f might reflect the expected reward or transition behavior. These encodings allow partition refinement to exploit structural symmetries and behavioral equivalences in a wide range of domains, especially if problems in that domain can be encoded as graphs. Further domains of application include Model Checking [3] and sorting algorithms [5].

Furthermore, if the underlying graph is bipartite, the splitter-function is expressing a vertex property such as Function 2.4 or 2.5 and each vertex on one of the two sides of the graph is in its own class, then the partition refinement algorithm can be implemented more efficiently as highlighted in [6]. This is due to the fact that one of the sides is already fully refined, thus only the other side might change. Combined with the fact that the graph is bi-partite, splits that occur within one cell cannot affect other cells. Thus, we don't have to "go back" during the execution of the loop in Algorithm 2.

Algorithm 2 More efficient refinement, if graph $G = ((U, V), E)$ bipartite and $\forall v \in U : |E^{-1}(v)| \leq 1$ or $\forall v \in V : |E^{-1}(v)| \leq 1$. For the algorithm we assume the former.

Input: Initial partition $\Pi_{init} = \{A_1, A_2, \ldots, A_k\}$, Bi-partite graph $G = ((L, R), E)$
Output: Coarsest stable partition

```
function refineFast($\Pi_{init}, G$)
    $i \leftarrow -1$
    $\Pi_0 \leftarrow \Pi_{init}$
    for $v \in R$ do
        $i \leftarrow i + 1$
        $\Pi_i \leftarrow$ Refine($\Pi_{i-1}, E^{-1}(v)$)
    end for
    return $\Pi_i$
end function

function Refine($\pi, S$)
    for $A_i \in \pi$ do
        $\pi \leftarrow \pi \setminus A_i$
        $\pi \leftarrow \pi \cup \{A_i \cap S\} \cup \{A_i \setminus S\}$        ▷ Set $S$ "splits" $A_i$ into two parts
    end for
end function
```

2.5 Surprise and Entropy

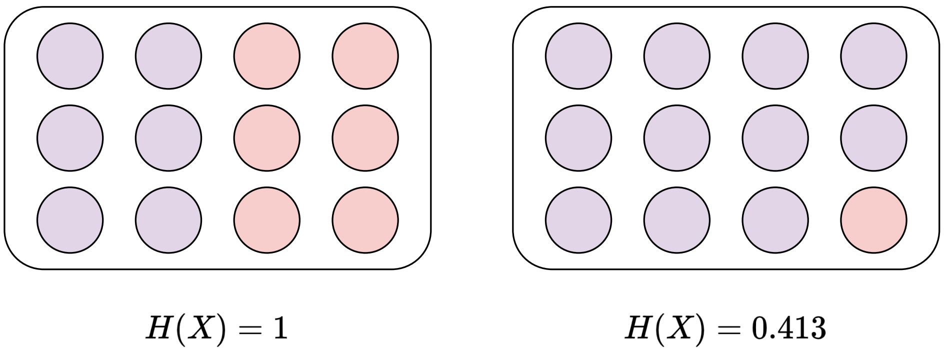


Figure 2.3: Entropy is measure of “surprise” and it increases with decreasing probability. On the left side, both colors are evenly distributed, so drawing either one is equally surprising. On the right side, drawing a red ball from the set of elements would be very surprising, because the probability is only $\frac{1}{12}$. But because this event is so unlikely, one does *not expect* to be surprised. As a result, the expected surprise - that is, the entropy - is low.

The *information value* or *surprisal* of an event E is defined as

$$I(E) = \log_b \left(\frac{1}{p(E)} \right) = -\log_b \left(p(E) \right) \tag{2.6}$$

It increases as the probability of the event $p(E)$ decreases. Intuitively, if the probability is close to 1, then one wouldn't be surprised if this event actually occurred, so the surprisal is close to 0.

The *entropy*, or *expected surprise*, $H(X)$ of a discrete random variable X which takes values in the set $\mathcal{X}$ is defined by equation 2.7 [7].

$$H(X) = \sum_{x \in \mathcal{X}} p(x) I(X) = -\sum_{x \in \mathcal{X}} p(x) \log_b p(x) \tag{2.7}$$

where $p(x) := \mathbb{P}[X = x]$.

If not specified any further, the base b of the logarithm is assumed to be 2. In chapter these concepts will be used to define a heuristic scoring system based on constraint names. ref

2.6 Adjusted Rand Index

The *Rand Index* is a statistical measure used to compare two different partitions $\pi = \{A_1, A_2, \dots, A_k\}, \pi' = \{B_1, B_2, \dots, B_l\}$ of elements from the same set $U = \{1, 2, \dots, n\}$. Let $f_\pi(x) : U \mapsto \mathbb{N}$ be a function mapping an element $x \in U$ to the index of its cell in partition π. Function $f_{\pi'}$ is defined analogously. Furthermore, let

$$E_{\circ_1,\circ_2} = \{(x,y) \in U \times U \mid (f_\pi(x) \circ_1 f_\pi(y)) \wedge (f_{\pi'}(x) \circ_2 f_{\pi'}(y))\}$$

Intuitively, e.g. the set $E_{=,\neq}$ refers to the set of pairs $(x,y) \in U \times U$ of elements which are in the same cell in π, but in different cells in π'. Now we can define the *Rand Index* as follows:

$$\mathrm{RI} = \frac{\overbrace{|E_{=,=}| + |E_{\neq,\neq}|}^{\text{Number of pairs for which } \pi,\pi' \text{ agree}}}{\underbrace{|E_{=,=}| + |E_{\neq,=}| + |E_{=,\neq}| + |E_{\neq,\neq}|}_{\text{Number of all pairs}}} = \frac{|E_{=,=}| + |E_{\neq,\neq}|}{\binom{n}{2}} \quad \in [0,1] \tag{2.8}$$

With appropriate data structures, e.g. a mapping between elements and cell index for each partition, Equation 2.8 can be evaluated in $O(n)$.

The *Adjusted Rand Index* is a chanced-adjusted version of the regular *Rand Index* which accounts for similarities that might occur by random chance. It is one of the most popular measures for comparing partitions or clusters and can be computed by using Equation 2.9 [8].

$$\mathrm{ARI} = \frac{\mathrm{RI} - \text{Expected RI}}{\text{Max RI} - \text{Expected RI}} = \frac{\sum_{ij}\binom{v_{ij}}{2} - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \div \binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \div \binom{n}{2}} \tag{2.9}$$

The values v_{ij}, a_i and b_j are taken from the so called Contingency Table shown in Figure 2.1.

The notation of Table 2.1 makes it seem like that $l \cdot k$ set intersection operations have to be computed in order to compute the full contingency table. Computing the intersection $A \cap B$, with $A, B \subseteq U$ being of roughly similar size, can be quite an expensive operation, depending on the precise data structures used. However, when there is an efficient data structure available mapping each element to the index of its containing cell, then the contingency table can be computed in one pass over all elements as shown in Algorithm.

$\pi' \setminus \pi$	A_1	A_2	$\dots$	A_k	**sums**
B_1	v_{11}	v_{12}	$\cdots$	v_{1k}	a_1
B_2	v_{21}	v_{22}	$\cdots$	v_{2k}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
B_l	v_{l1}	v_{l2}	$\cdots$	v_{lk}	a_l
sums	b_1	b_2	$\cdots$	b_k	

Table 2.1: Contingency Table of partitions π and π'. Entry v_{ij} denotes the number of elements sets A_i and B_j have in common, i.e., $v_{ij} = A_i \cap B_j$.

Algorithm 3 A simple algorithm computing the contingency table with one pass over all elements of U.

Input: Partitions $\pi = \{A_1, A_2, \dots, A_k\}, \pi' = \{B_1, B_2, \dots, B_l\}$, function $f_C : U \mapsto \mathbb{N}$ for $C = \{C_1, C_2, \dots\} \in \Pi(U)$ mapping $u \in U$ to C_i iff $u \in C_i$

Output: Contingency Table of partitions π, π' and sums of columns and rows

```
function COMPUTECONTINGENCYMATRIX($\Pi_{init}, G$)
    $A \leftarrow$ 2D Array with $l$ rows and $k$ columns
    $sumsOfColumns \leftarrow$ 1D Array with $k$ entries
    $sumsOfRows \leftarrow$ 1D Array with $l$ entries
    for $u \in U$ do
        $column \leftarrow f_\pi(u)$
        $row \leftarrow f_{\pi'}(u)$

        $A[row, column] \leftarrow A[row, column] + 1$
        $sumsOfColumns[column] \leftarrow sumsOfColumns[column] + 1$
        $sumsOfRows[row] \leftarrow sumsOfRows[row] + 1$
    end for
    return $(A, sumsOfRows, sumsOfColumns)$
end function
```

We refer to [9] for a more detailed discussion on different similarity measures and their mathematical derivation.

3 Related Work

4 Generic Column Generation (GCG)

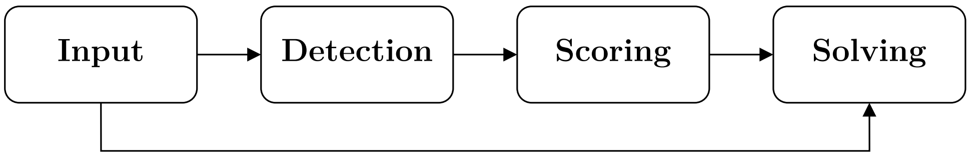


Figure 4.1: A simplified overview of the four major stages of solving a model with GCG.

In this chapter, we introduce Generic Column Generation (GCG), a decomposition solver which is based on the open-source MIP-Solver SCIP (SCIP) [10]. Readers already experienced with GCG and its capabilities may still find some details and observations interesting. For a given problem, GCG is able to perform an automatic Dantzig-Wolfe reformulation which is then solved using a branch-price-and-cut algorithm. Alternatively, GCG support a special *Benders-Mode* which reformulated the problem using Benders decomposition.

In contrast to other open-source solvers like BaPCod (Branch-and-price Code) [11] or commercial software such as *SAS* [12] which rely solely on user-provided decompositions, GCG is able to automatically detect different kinds of structures algorithmically, including but not limited to

- Single-Bordered structures
- Arrowhead structures using the third-party tool hMETIS (Hypergraph METIS) [13].
- Staircase structures

Entfernen und auf Kapitel vorher ref

The solving process in divided into multiple consecutive stages as shown in Figure 4.1. Each stage will be explained in more detail in the following section as needed. The detection in particular aims to make GCG more accessible to a wider range of users which do not necessarily have the required theoretical background and practical experience to reformulate linear programs on their own. For more details about individual features and capabilities, we refer to the official documentation [14].

Kurz die 4 Schritte aus Bild erwähnen und einen Satz

4.1 Detection

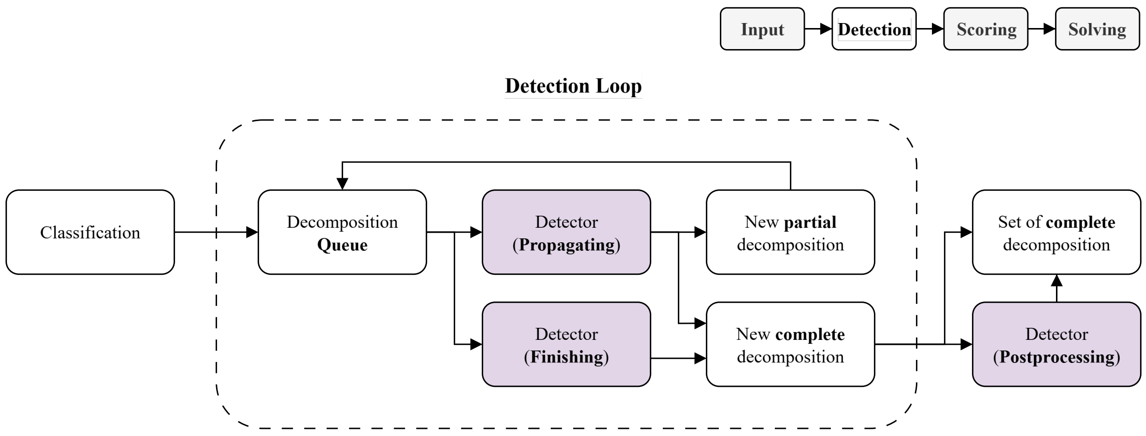


Figure 4.2: A simplified overview of the detection process and its detection loop.

As mentioned in the introduction to this chapter, one integral part and distinguishing feature of GCG is its detection framework. A simplified overview of the detection currently [1] implemented in GCG is shown in Figure 4.2. For a more detailed visualization including additional information about how pre-solving is handled we refer to the official documentation [14]. The framework consists of two major parts:

1. A **classification** step, in which a set of classifiers is partitioning the constraints (and variables) according to a certain property, producing one partition each. The goal of this step is to detect different underlying structures of the constraint matrix, which can be used during the detection loop to make more informed decisions about which constraints to assign to which block or master. Important classifiers for the remainder of this thesis are discussed in more detail in Section 4.2.

2. The **detection loop**, which consists of a set of detectors which are responsible for assigning constraints either the master or to individual blocks. In round $n+1$ a detector receives a *partial* decomposition, that is, a decomposition in which *not all* constraints are assigned yet, from round n as input and pushes a set of newly created (partial) decompositions to a queue. In case the user did not provide a partial decomposition as input in round 0, the loop is initialized with a decomposition in which no constraint is assigned yet.

[1]GCG version 3.5, as of 2025-07-18.

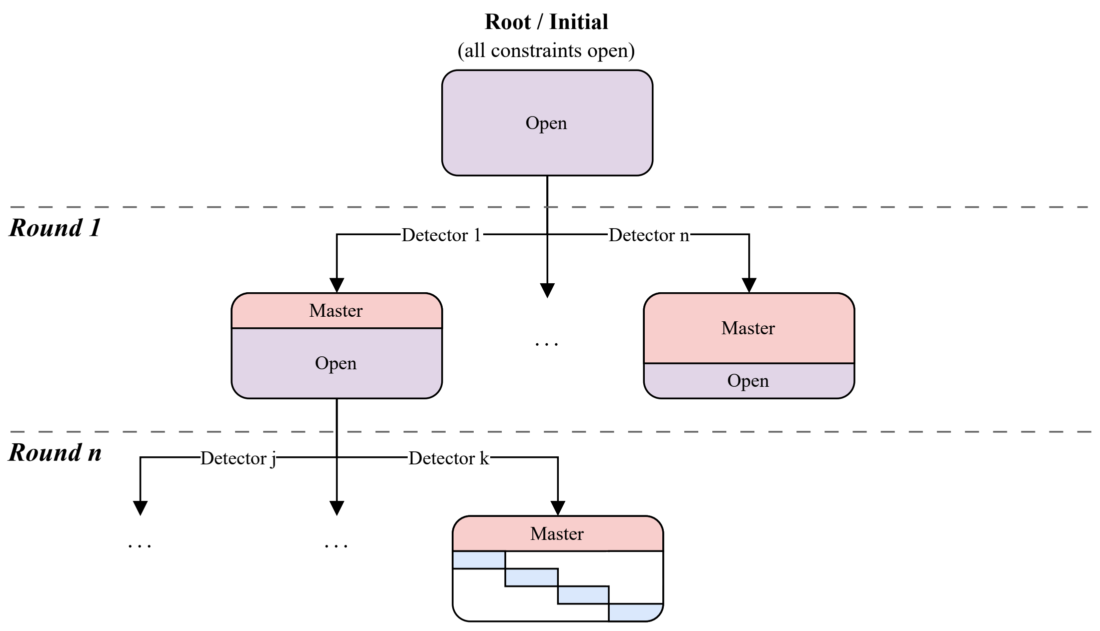


Figure 4.3: Visualization of the induced tree of propagated partial decompositions.

The concept of detecting structures in different rounds is visualized in Figure 4.3. Starting from a root decomposition in which all constraints are still unassigned or "open", different detectors produce a set of new partial decomposition. Depending on the configuration, a detector is not allowed to work on a certain partial decomposition or its decedents twice. A very simple but concrete example of how such a tree might look like in practice can be found in Section 4.4.

Furthermore, if no detector found any new decomposition in round k, or k exceed the maximum number of rounds, the detection loop is stopped and all complete decomposition are collected, scored and exactly one is chosen for which the solving is started. The scoring and selection stage is of particular interest in practice, because the tree in Figure 4.3 might grow beyond thousand of decompositions, of which the best in terms of solving time or a different metric must be selected. Because the scoring of decompositions is not of major interest for *this* thesis, we refer to the official documentation for details [14].

Grammati Wort- wahl

4.2 Classifiers

As mentioned in the introduction to this chapter, classifiers are responsible for detecting different underlying structures of the constraint matrix, which can be used during the detection loop to make more informed decisions about which constraints to assign to which block or master. Given a set of constraints $C = \{c_1, c_2, \ldots, c_m\}$, classifiers can be seen as a *injective* function $f : C \mapsto \mathbb{Z}$, i.e., a function that assigns each constraint to exactly one number or *class*. Note that in GCG, classifiers are allowed to only classify a subset $C' \subseteq C$, leaving $C \setminus C'$ unassigned to any class [2]. In the current version of GCG, however, all classifiers always assign every constraint to some class.

4.2.1 Name Classifiers

The names of constraints and variables are, if provided, a strong indicator to which constraints or variables are related to each other. The names usually consist of two parts:

1. The semantic group name, such as "capacity" or "link" for e.g. a Bin-Packing model.
2. A *modifier*, which usually consists of numbers, capital letters or a combination of both. Typically, the modifier is separated from the semantic group name via. non alpha-numeric characters such as "_" or "#".

Constraints in the same group typically share similar names, with the *modifier* being the only differentiating factor. For example, in a Bin-Packing problem, capacity constraints such as "capacity_1", "capacity_2", ... usually vary only in the index indicating the bin. This similarity can be quantified using metrics like the *Levenshtein Distance*, which is the minimum number of single-character edits required to change one word into the other.

Because there is no standardized naming scheme for either variables or constraints, name classifiers usually operate under the assumption that the modeler provided *reasonable* names, if any at all. A non-exhaustive collection of different naming schemes observed is provided in Appendix .

Appendix

[2]When using GCG as a library, this can be checked via. `IndexPartition::isIndexClassified`.

Given a alphabet Σ, words $w, v \in \Sigma^*$ over that alphabet, then the *Levenshtein* distance between those two words can be computed as:

$$\text{lev}(w,v) = \begin{cases} |w| & \text{if } v = \epsilon \\ |v| & \text{if } w = \epsilon \\ \text{lev}(\text{prefix}(w), \text{prefix}(v)) & \text{if } \text{head}(w) = \text{head}(v) \\ 1 + \min \begin{cases} \text{lev}(\text{prefix}(w), v) \\ \text{lev}(w, \text{prefix}(v)) \\ \text{lev}(\text{prefix}(w), \text{prefix}(v)) \end{cases} & \text{otherwise} \end{cases} \tag{4.1}$$

Equation 4.1 can be computed in $O(|w| \cdot |v|)$ using a dynamic programming approach. Let $B = (\text{lev}(\text{name}(c_i), \text{name}(c_j)))_{1 \le i,j \le m}$ the pair-wise Levenshtein Distance between constraint names, $k \in \mathbb{N}$ the *connectivity* and $G = (V, E)$ with $V = \{c_1, c_2, \dots, c_m\}, E = \{\{u, v\} \mid u, v \in V, u \ne v, \text{lev}(\text{name}(u), \text{name}(v)) \le k\}$. Furthermore, let $reach(v)$ be the set of reachable vertices from vertex $v \in V$ which is defined as the fix-point of the following function for $s = v$:

$$reachEventually_t(T) = \{u \in V \mid \exists v \in T : v \in E(u)\ \} \cup \{t\}$$

Then two constraints $c_i, c_j \in V$ are in the same class iff $c_j \in reach(c_i)$. A small example of this concept is shown in Figure 4.4. This idea can also be applied to variable names. 5000 cons limit

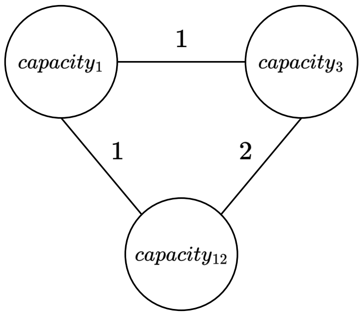


Figure 4.4: The graph of pair-wise Levenshtein weights for three capacity constraints. For $k = 1$, the edge between $capacity_3$ and $capacity_{12}$ vanishes, but because they is still a connecting path via. $capacity_1$, both constraints are assigned to the same class.

4.2.2 Numeric Classifiers

Nonzero

$$A = \begin{array}{c} \\ cons_1 \\ cons_2 \\ cons_3 \\ cons_4 \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{pmatrix} 1 & 5 & -1 & -1 \\ 20 & 0 & 0 & 20 \\ 20 & 10 & 10 & 0 \\ 0 & 100 & -100 & 100 \end{pmatrix} \end{array} \begin{array}{c} class \\ 4 \\ 2 \\ 3 \\ 3 \end{array}$$

Figure 4.5: A constraint matrix with coefficients for each variable. Each constraint is assigned to a class corresponding to its number of non-zero entries.

The nonzero classifier classified constraints according to their number of non-zero variable coefficients as shown in Figure 4.5. Many types of models including Bin-Packing and Cutting-Stock consist only of constraint groups with a rather "stable" internal structure, i.e., the capacity constraint for each bin in model 4.4 consist of the same number of variables, because each constraint is just a sum over all items differing only in index for the respective bin. In general, constraint groups that are suited for this type of classifiers usually involve summations over fixed-sized sets (e.g. a set of items or bins) whose choice is not dependent on any quantified variable. Example for the latter include problems whose formulation is based on graphs and usually contains flow-conservation constraints shown in Equation 4.2.

Wrong ref

$$\sum_{u \in E(v)} x_{uv} - \sum_{u \in E^{-1}(v)} x_{vu} = 0 \quad \forall v \in V \tag{4.2}$$

The amount of non-zeros in these constraint is entirely dependent on the number of outgoing and incoming edges for each vertex.

Objective Function

A simple classification for variables can be done using information from the objective function, such as:

1. Partition variables according to the sign of their coefficient in the objective function. This approach yields three classes in total Positive, Negative and Zero.
2. Partition them according to the actual *value* of the coefficient.

Partitioning variables according to the first approach is sufficient for models such as Bin-Packing, in which only the y-variables appear in the objective function.

The second approach might partition the variables in too many small cells when e.g. different costs are associated with variables in the objective function. This behavior can be observed on model types such as Multi-Commodity-Flow and Unit-Commitment.

4.2.3 Type Classifiers

Type classifiers examine the constraint matrix to infer a higher-level *type* for each individual constraint. A key objective of such classifiers is ensuring or at least improving *robustness*. Even minor modifications to a single constraint - such as the removal of one variable - can lead to a different classification, as seen with the previously discussed nonzero classifier. Moreover, the likelihood of such changes increases when pre-processing is enabled.

SCIP Types

When using GCG as a library, the type of a variable or constraint can be retrieved via. `SCIPconsGetType(cons)` or `SCIPvarGetType(cons)` respectively. The former function is not provided by SCIP itself, but is implemented in GCG instead. The implementation compares the name of the handler the constraint is assigned to and compares it to a known list of constraint handlers. The list of supported handlers includes *Knapsack*, *Set Partitioning*, *Set Covering*, *Set Packing*, *Varbound* and *General*, in case no special structure was detected. Variables can be classified as *Integer*, *Binary* or *Continuous* [3].

the

Check List

The clear downside of this classification is its important precondition. In order to use this feature properly and retrieve a meaningful type via. the two methods, pre-solving must have been executed prior to detection. When GCG reads the problem as e.g. an `.lp` file, all constraints are added as linear constraints to the underlying SCIP model. These constraints are usually “upgraded” if possible, that is, their structure is analyzed and assigned to the correct constraint handler during pre-solving. This is done in order to take advantage of properties only possessed by certain types of constraints, e.g. a solution to a set of Knapsack constraints *can* be computed more efficiently by using an algorithm based on dynamic programming. For more detailed information we refer to the official documentation [15].

[3]There are more types of variables in newer versions of SCIP such as *Implicit Integer*, but these three basic types are sufficient for the purpose of this discussion.

Preliminary testing showed that it is not trivial to configure the pre-processing in such a way that *only* the upgrade mechanism is triggered and variables and constraints remain unchanged.

Add test config to appendix

MIPLIB Constraint Types

Nr.	Type	Linear Constraint	Notes
1	Empty	$\varnothing$	-
2	Free	$-\infty \leq x \leq \infty$	No finite side.
3	Singleton	$a \leq x \leq b$	-
4	Aggregation	$ax + by = c$	-
5	Precedence	$ax - ay \leq b$	x, y have same type.
6	Variable Bound	$ax + by \leq c$	$x \in \{0,1\}$
7	Set Partitioning	$\sum 1x_i = 1$	$\forall i : x_i \in \{0,1\}$
8	Set Packing	$\sum 1x_i \leq 1$	$\forall i : x_i \in \{0,1\}$
9	Set Covering	$\sum 1x_i \geq 1$	$\forall i : x_i \in \{0,1\}$
10	Cardinality	$\sum 1x_i = b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
11	Invariant Knapsack	$\sum 1x_i \leq b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
12	Equation Knapsack	$\sum a_i x_i = 1$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
13	Bin Packing	$\sum a_i x_i + ay \leq a$	$\forall i : x_i, y \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
14	Knapsack	$\sum a_i x_i \leq b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
15	Integer Knapsack	$\sum a_i x_i \leq b$	$\forall i : x_i \in \mathbb{Z}, b \in \mathbb{N}$
16	Mixed Binary	$\sum a_i x_i + \sum p_j s_j \ \{\leq, =\}\ b$	$\forall i : x_i \in \{0,1\}, \forall j : s_j$ continuous
17	General Linear	$\sum a_i x_i \ \{\leq, \geq, =\}\ b$	No special structure.

Table 4.1: The structure of all 17 constraint types MIPLIB keeps track of.

In contrast to the automatic constraint classification performed by SCIP during presolving, the MIPLIB benchmark set provides its own static classification scheme [16]. This classification assigns constraints to a set of well-defined structural types such as knapsack, set-partitioning and others as shown in Table 4.1. Since it is based solely on the syntactic form of the constraints in the original model, it can be applied *independently* of solver presolving. Because all types shown in Table 4.1 are deducible only from *local* information such as type of variables and right hand side coefficient, the types can be detected with one pass over the constraint matrix.

This type of classifier shares some issues related to robustness with numeric classifiers. Constraint types such as Singleton, Aggregation or Variable Bound depend on the number of non-zeroes, leading to potential miss-classifications on graph based models. Furthermore, the only differentiating factor for more “complex” types such as *Bin Packing* and *Knapsack* is the presence of a variable which happens to have the same coefficient as the right-hand of that constraint.

4.3 Existing Detectors

$$
\begin{array}{llll}
\min & \sum_{j=1}^{m} y_j & & \\
\text{s.t.} & \sum_{j=1}^{m} x_{ij} = 1 & \forall i \in \mathcal{I} & (4.3) \\
& \sum_{i=1}^{n} a_i x_{ij} \leq C y_j & \forall j \in \mathcal{J} & (4.4) \\
& x_{ij} \in 0, 1 & \forall i \in \mathcal{I}, \forall j \in \mathcal{J} & \\
& y_j \in 0, 1 & \forall j \in \mathcal{J} &
\end{array}
$$

Figure 4.6: Bin-Packing Model with items $\mathcal{I} = \{1, \ldots, n\}$, item sizes $a_i \in \mathbb{Z}_{\geq 0}$, bins $\mathcal{J} = \{1, \ldots, m\}$ and capacity C.

Bild

Nr.	Master	Open
1	(4.3)	(4.4)
2	(4.4)	(4.3)
3	(4.3), (4.4)	-

Table 4.2: For each classifier, the *cons class* detector will produce $2^k - 1$ new partial decompositions with k being the number of classes.

4.4 Example

In order to illustrate the detection with a concrete example, we revisit the textbook Bin-Packing model shown in Figure 4.6. Constraints 4.3 enforce that every item is packed in exactly one bin, while inequalities 4.3 ensure that the capacity of each bin is respected if some item is packed in it. The objective is to minimize the number of bins.

Without pre-solving enabled, a classifier such as MIPLIB would assign constraints 4.3 and 4.4 to the classes *Set Partitioning* and *Bin-Packing* respectively. If unique, this classification is added to a list provided to the detection stage.

If no further classifications are found, the *cons class* detector will yield 3 new partial decompositions, first assigning constraints 4.3, then 4.4 and finally both 4.3 and 4.4 to the master. The constraint group not assigned to the master remains *open*.

During the next round of detection, a detector such as *Connected Base* will receive the partial decomposition with only the packing constraints assigned to the master as input. Here, the induced constraint adjacency graph of the $q \geq 0$ open capacity constraints consists of q isolated connected components, forming the desired block-diagonal structure.

5 Tree Refinement

With the existing capabilities of GCG presented in the previous chapter, we continue with the main contributions of this thesis:

- A new module which is integrated into the detection framework of GCG for reverse engineering semantic groupings of the original formulation.
- Additional auxiliary classifiers which implement constraint and variable classification based on information not currently used including examples of *when* they are crucial detecting semantics.

This chapter is divided into three main section:

1. A short summary about the available information we have access to.
2. What the motivation and goals are why and how we aim to process this information.
3. The concrete algorithm and its most integral parts.

Some concrete details about the implementation itself are not subject of the following sections, but are discussed in Chapter 6.

5.1 Information

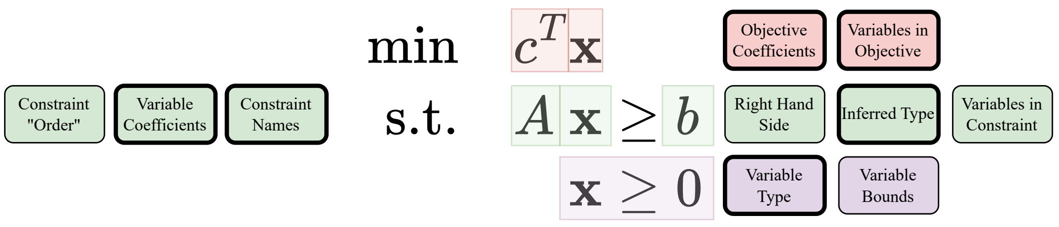


Figure 5.1: All parts of a model that contain useful information for semantic grouping of constraints and variables. Elements with a thick border are already used as a key concept in one of the existing detectors.

Before we present any algorithmic details, we give an overview about the available information.

1. **Objective**: For the objective functions, information about the participating variables and their coefficients is available. For some models e.g. for Bin-Packing, this information alone is sufficient to partition the variables.

2. **Coefficients**: The use of coefficients to classify constraints and variables was already in discussed in Section 4.2.

3. **Bounds**: For all variables $lb \leq x \leq ub$ information about their lower- and upper-bounds is available. Furthermore, the left- and right-hand-side of linear constraints $lhs \leq \sum_i a_i x_i \leq rhs$ are available as well.

4. **Types**: Variable types such as *Integer* are usually stated explicitly in the input format. If not, then information about the variable bounds can be used to deduce a type, e.g. $0 \leq x \leq 1$ is a strong indicator that x is a *Binary* variable.

5. **Names**: If specified by the modeler, then variables and constraints might have meaningful names which can be used as a strong indicator which constraints and variables belong to the same group.

6. **Order**: In contrast to other kinds of information, the constraint “order” is no intrinsic property of the model itself. With the term “order”, we refer to the order of the constraints as specified in the input format. When a model is created e.g. via. a script, constraints are usually added in *blocks* by the modeler. This information is used in Section 5.3.3 to conceptualize a classifier based on that.

5.2 Motivation

5.3 Classifiers

In the following Sections we describe different classifiers which are not yet implemented in GCG but could potentially provide new information about the model.

5.3.1 Bounds

When considering bounds, we differentiate between two types:

1. Bounds of a single variable lower bound $\leq x \leq$ upper bound
2. Constraint Bounds, lhs $\leq \sum a\mathbf{x} \leq$ rhs

Variable bounds

Constraint bounds

5.3.2 Relaxed MIPLIB types

Nr.	Type	Linear Constraint	Notes
1	Empty	$\varnothing$	-
2	Free	$-\infty \le x \le \infty$	No finite side.
3	Singleton	$a \le x \le b$	-
4	Aggregation	$ax + by = c$	-
5	Precedence	$ax - ay \le b$	x, y have same type.
6	Variable Bound	$ax + by \le c$	$x \in \{0,1\}$
7	Set Partitioning	$\sum 1x_i = 1$	$\forall i : x_i \in \{0,1\}$
8	Set Packing	$\sum 1x_i \le 1$	$\forall i : x_i \in \{0,1\}$
9	Set Covering	$\sum 1x_i \ge 1$	$\forall i : x_i \in \{0,1\}$
10	Cardinality	$\sum 1x_i = b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\ge 2}$
11	Invariant Knapsack	$\sum 1x_i \le b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\ge 2}$
12	Equation Knapsack	$\sum a_i x_i = 1$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\ge 2}$
13	Bin Packing	$\sum a_i x_i + ay \le a$	$\forall i : x_i, y \in \{0,1\}, b \in \mathbb{N}_{\ge 2}$
14	Knapsack	$\sum a_i x_i \le b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\ge 2}$
15	Integer Knapsack	$\sum a_i x_i \le b$	$\forall i : x_i \in \mathbb{Z}, b \in \mathbb{N}$
16	Mixed Binary	$\sum a_i x_i + \sum p_j s_j \; \{\le, =\} \; b$	$\forall i : x_i \in \{0,1\}, \forall j : s_j$ continuous
17	General Linear	$\sum a_i x_i \; \{\le, \ge, =\} \; b$	No special structure.

Table 5.1: The structure of all 17 constraint types MIPLIB keeps track of.

This table

5.3.3 Ordered Voting

Ordered Voting

Co-occurence clustering

Co

5.4 Strategies

Strategies are *the* central building block of the algorithm and are responsible for refining sets of constraints or variables. Let U be a the total set of constraints or variables. Each strategy gets a single set $S \subseteq U$ as input and produces a partition $\pi_S \in \Pi(S)$ of that set as output. Each set $A_i \in \pi_S$ corresponds to one child node. Conceptually, each strategy can be seen as a materialization of a specific splitter function as defined in Section 2.4. In the following, we differentiate between re-callable and non re-callable strategies. The former type may be called more than once in any given sub-tree, as the result may depend on the set that is being refined.

5.4.1 Slice (Partition)

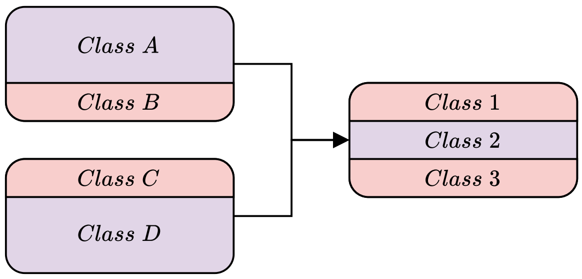


Figure 5.2: A simplified illustration assuming that constraint in both partitions can be rearranged into continuous blocks, thus "slicing" the partitions into different-sized chunks. The output partition inherits all these slices.

Slicing strategies are the most simple type of strategy. Given two partition $\pi, \pi' \in \Pi(U)$, we define the *combined partition* $\pi \sqcap \pi'$ as follows:

$$\pi \sqcap \pi' = \{A_i \cap B_j \mid A_i \in \pi, B_j \in \pi'\} \setminus \varnothing \tag{5.1}$$

This concept is illustrated in Figure 5.2. Because strategies only refine single sets and not whole partitions, Equation 5.1 always degenerates to $\pi_{slice} \sqcap \{S, U \setminus S\}$ for some partition $\pi_{slice} \in \Pi(U)$ and a set S we want to refine. The result is then restricted to elements of S, which yields a partition of S. More formally, this strategy can be expressed as Function 5.2 and computed efficiently using Algorithm 4.

$$f(\pi_{splitter}, S) = \{A_i \cap S \mid A_i \in \pi_{splitter}\} \tag{5.2}$$

Algorithm 4 If a lookup table represented by function f is available, then Function 5.2 can be implemented in $\mathrm{O}(|S|)$.

Input: Partition $\pi = \{A_1, A_2, \ldots, A_k\}$, set $S \subseteq U$, function $f_C : U \mapsto \mathbb{N}$ for $C = \{C_1, C_2, \ldots\} \in \Pi(U)$ mapping $u \in U$ to C_i iff $u \in C_i$
Output: Partition of S according to Function 5.2.

```
function strategySlice($\pi, S$)
    $\pi_{out} \leftarrow$ list of $k$ empty sets $B_1, B_2, \ldots, B_k$
    for $s \in S$ do
        $i \leftarrow f_S(s)$
        $B_i \leftarrow B_i \cup \{s\}$
    end for
    Remove empty sets from $\pi_{out}$
    return $\pi_{out}$
end function
```

5.4.2 Slice (Covering)

This strategy is functionally equivalent to the refinement method “fast” from [6]. The name most likely stems from an algorithm informally described a fast algorithm based on bucket sort to implement such a partition refinement algorithm.

5.4.3 Recursive

The *recursive* strategy is the the only strategy which utilizes the full blown partition refinement framework.

5.5 Scoring

5.5.1 Constraint Names

Algorithm 5

Input: Name of a constraint
Output: Relevant semantics of the constraint name if the name adheres

```
function EXTRACTSEMANTICPART($name$)
    $name_{new} \leftarrow name$
    $name_{old} \leftarrow name$
    repeat
        $name_{old} \leftarrow name$
        $start \leftarrow$ Position of opening character e.g. [, {, (, ...
        $end \leftarrow$ Position of corresponding closing character e.g. ], }, ), ...
    until $name_{new} = name_{old}$
end function
```

WIP

5.5.2 Ground Truth based

Ground-Truth

5.5.3 Connected Block Score

Connected

6 Implementation

6.1 Architecture

6.2 Metadata

6.3 Multi-Threading

6.4 Hashing

6.5 Cutoff Conditions

6.5.1 Local Conditions

6.5.2 Global Conditions

7 Evaluation

7.1 Setup

Type	**Name**	**Metric**
CPU	AMD Ryzen 3700X	3.8 GHz
RAM	-	16GB

Table 7.1: Consumer-grade components used to run all experiments.

All experiences were run on a system with components as specified in Table 7.1.

7.2 StrIPlib

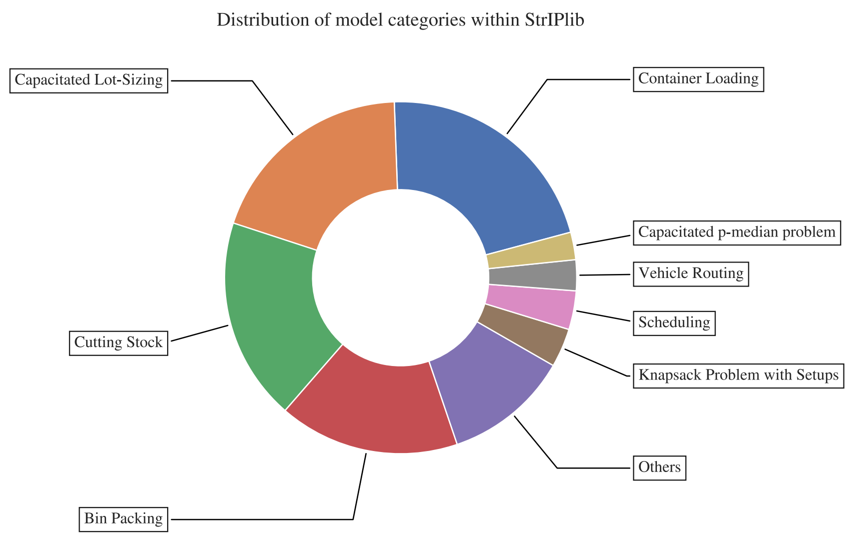


Figure 7.1: The distribution of different categories of model within strIPlib. Most problems are part of "common" categories like Bin-Packing, Scheduling and Routing. The category "Others" includes e.g. Fantasy Football, Train Scheduling and different types for which only a small number of model files are available.

The Structured Integer Program Library (strIPlib) is a collection of over 21000 mixed-integer programs with an exploitable structure such as Block-Diagonal and Staircase . All instances are assign to exactly one of the 33 main categories as highlighted in Figure 7.1. Each categories is further sub-divided into a number of smaller sub-categories, because each kind of problem can be modeled (e.g. three-index vs. four-index) *or* decomposed in a variety of different ways.

ref

The number of available instances per category ranges from as low as 2 for Binary/Ternary Code Construction and up to ≈ 4700 for Container Loading, which makes the data-set in-balanced with respect to available models per main category. This is only of theoretical concern and is further discussed in section . Furthermore, the largest four categories account for ≈ 80% of the total instance count with the remaining 20% distributed across 29 categories. One singular category "Haplotype Inference" with 40 instances is excluded from all tests, because the problem files are not readable by either GCG or SCIP. This behavior can be traced back to the used variable names in these models, which all contain the special character "^".

ref

Models ohne Namen noch ergänzen (Prob- lem-

7.3 Stuff

8 Notes

8.1 MIPLIB Constraint Types

8.2 Relaxed Constraint Types

Nr.	**Type**	**Linear Constraint**	**Notes**
1	Empty	$\varnothing$	-
2	Free	$-\infty \leq x \leq \infty$	No finite side.
6	Variable Bound	$ax + by \leq c$	$x \in \{0,1\}$
7	Set Partitioning	$\sum 1x_i = 1$	$\forall i : x_i \in \{0,1\}$
8	Set Packing	$\sum 1x_i \leq 1$	$\forall i : x_i \in \{0,1\}$
9	Set Covering	$\sum 1x_i \geq 1$	$\forall i : x_i \in \{0,1\}$
10	Cardinality	$\sum 1x_i = b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
10	Cardinality	$\sum 1x_i = b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
11	Invariant Knapsack	$\sum 1x_i \leq b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
12	Equation Knapsack	$\sum a_i x_i = 1$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
13	Bin Packing	$\sum a_i x_i + ay \leq a$	$\forall i : x_i, y \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
14	Knapsack	$\sum a_i x_i \leq b$	$\forall i : x_i \in \{0,1\}, b \in \mathbb{N}_{\geq 2}$
15	Integer Knapsack	$\sum a_i x_i \leq b$	$\forall i : x_i \in \mathbb{Z}, b \in \mathbb{N}$
16	Mixed Binary	$\sum a_i x_i + \sum p_j s_j \ \{\leq, =\}\ b$	$\forall i : x_i \in \{0,1\}, \forall j : s_j$ continuous
17	Mixed	$\sum a_i x_i \ \{\leq, \geq\}\ b$	No special structure.
17	Mixed Equation	$\sum a_i x_i \ \{=\}\ b$	No special structure.

Table 8.1: A relaxed version of the constraint types MIPLIB uses.

8.3 Classifiers, Detectors

Classfiers:

- VarTypes, ConsTypes (SCIP)
- MIPLIB Cons
- Name based / Levenstein
- Non-Zeroes
- Objective Values / Sign

Detectors:

- Cons Class, Var Class
- Connected Base
- HMETIS
- Dense Master Cons
- Detectors for single constraint types
- Staircase Heur
- Greedy
- Post Process

Literaturverzeichnis

[1] *Branch & Price.*

[2] "AN n Log n ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON". In: Hopcroft, J. *Theory of Machines and Computations.* Elsevier, 1971, pp. 189–196. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780124177505500221 (visited on 07/20/2025).

[3] Baier, C./ Katoen, J.-P. *Principles of Model Checking.* Cambridge, Mass: The MIT Press, 2008. 975 pp.

[4] Paige, R./ Tarjan, R. E. "Three Partition Refinement Algorithms". In: *SIAM Journal on Computing* 16.6 (12/1987), pp. 973–989. URL: http://epubs.siam.org/doi/10.1137/0216062 (visited on 08/19/2025).

[5] Mehlhorn, K./ Sanders, P. *Algorithms and Data Structures: The Basic Toolbox.* Berlin: Springer, 2008. 300 pp.

[6] Salvagnin, D. "Detecting Semantic Groups in MIP Models". In: *Integration of AI and OR Techniques in Constraint Programming.* Ed. by Quimper, C.-G. Vol. 9676. Cham: Springer International Publishing, 2016, pp. 329–341. URL: http://link.springer.com/10.1007/978-3-319-33954-2_24 (visited on 02/02/2025).

[7] Cover, T. M./ Thomas, J. A. *Elements of Information Theory.* 2nd ed. Hoboken, N.J: Wiley-Interscience, 2006.

[8] Sundqvist, M./ Chiquet, J./ Rigaill, G. *Adjusting the Adjusted Rand Index – A Multinomial Story.* 11/17/2020. arXiv: 2011.08708 `[stat]`. URL: http://arxiv.org/abs/2011.08708 (visited on 07/31/2025). Pre-published.

[9] Warrens, M. J./ Van Der Hoef, H. "Understanding the Adjusted Rand Index and Other Partition Comparison Indices Based on Counting Object Pairs". In: *Journal of Classification* 39.3 (11/2022), pp. 487–509. URL: https://link.springer.com/10.1007/s00357-022-09413-z (visited on 07/31/2025).

[10] “Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs”. In: Gamrath, G./ Lübbecke, M. E. *Lecture Notes in Computer Science.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 239–252. URL: http://link.springer.com/10.1007/978-3-642-13193-6_21 (visited on 07/08/2025).

[11] Sadykov, R./ Vanderbeck, F. *BaPCod - a Generic Branch-and-Price Code.* Technical Report. Inria Bordeaux Sud-Ouest, 11/2021. URL: https://inria.hal.science/hal-03340548.

[12] *SAS: Data and AI Solutions.* URL: https://www.sas.com/en_us/home.html (visited on 07/08/2025).

[13] Karypis, G. et al. “Multilevel Hypergraph Partitioning: Application in VLSI Domain”. In: *Proceedings of the 34th Annual Conference on Design Automation Conference - DAC ’97.* The 34th Annual Conference. Anaheim, California, United States: ACM Press, 1997, pp. 526–529. URL: http://portal.acm.org/citation.cfm?doid=266021.266273 (visited on 07/08/2025).

[14] *GCG.* URL: https://gcg.or.rwth-aachen.de/ (visited on 07/08/2025).

[15] *SCIP Doxygen Documentation: Overview.* URL: https://www.scipopt.org/doc/html/ (visited on 07/08/2025).

[16] Gleixner, A. et al. “MIPLIB 2017: Data-driven Compilation of the 6th Mixed-Integer Programming Library”. In: *Mathematical Programming Computation* (2021). URL: https://doi.org/10.1007/s12532-020-00194-3.