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 July 21, 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

2.1 Linear Optimization

Linear Programming (LP) is a mathematical optimization technique used to determine the best possible outcome in a given model, whose requirements are represented by linear relationships. The goal is typically to maximize or minimize a linear objective function, subject to a set of linear equality and/or inequality constraints.

In a standard form, a linear programming problem can be expressed as follows:

$$\max c^T x$$

- x is the vector of decision variables,
- c is the vector of coefficients in the objective function,
- A is a matrix representing the coefficients of the constraints, and
- b is the right-hand side vector of the constraints.

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.

2.2 Dantzig-Wolfe Decomposition

2.3 Graph Theory

2.4 Partition Refinement

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

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

A partition $\pi = \{A_1, A_2, \ldots, A_k\}$ of a set U is called a refinement of a partition $\pi' = \{B_1, B_2, \ldots, B_m\}$, 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. Given a splitter-function $f : U \mapsto Q$ which maps every element of U to some element of an arbitrary target set Q, an initial partition π_{init} the goal is typically to find the coarsest partition $\pi_f = \{A_1, A_2, \ldots, A_k\}$ of U such that the following two properties hold:

1. The partition π_f is a refinement of the initial partition π_{init}
2. $\forall A_i \in \pi_f \; \forall a, b \in P_i : f(a) = f(b)$, that is, the function f can intuitively be thought of a function expressing a certain "property" for each element. Elements with different properties cannot be part of the same cell and must be separated from each other during the refinement process.

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.

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:

$$f_{\text{count}}(v, X_{\text{ind}}) = |\{v' \in V \mid \forall (v, v') \in E, v' \in X_{\text{ind}}\}| \tag{2.1}$$

$$f_{\text{exists}}(v, X_{\text{ind}}) = \begin{cases} 1 & |\{v' \in V \mid \forall (v, v') \in E, v' \in X_{\text{ind}}\}| > 0 \\ 0 & \text{else} \end{cases} \tag{2.2}$$

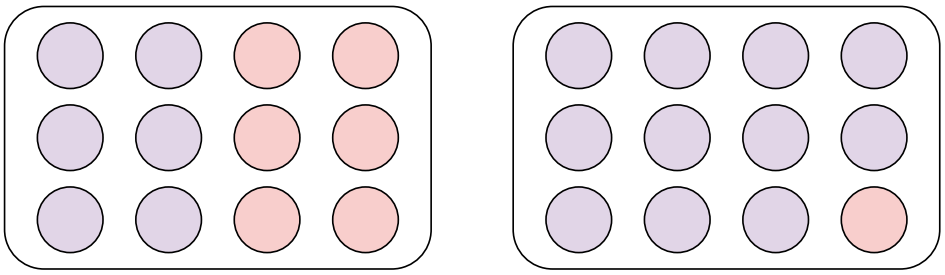
Functions 2.1 and 2.2

Algorithm 1 Partition refinement algorithm

while Test **do**
end while

Furthermore, if the underlying graph is bipartite and the splitter-function is expressing a vertex-property, i.e.,

2.5 Surprise and Entropy



$H(X) = 1$ $H(X) = 0.413$

Figure 2.1: 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

$$I(E) = \log_b\left(\frac{1}{p(E)}\right) = -\log_b\left(p(E)\right) \tag{2.3}$$

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 2.4 [1].

$$H(X) = \sum_{x \in \mathcal{X}} p(x) I(X) = -\sum_{x \in \mathcal{X}} p(x) \log_b p(x) \tag{2.4}$$

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

3 Related Work

4 GCG

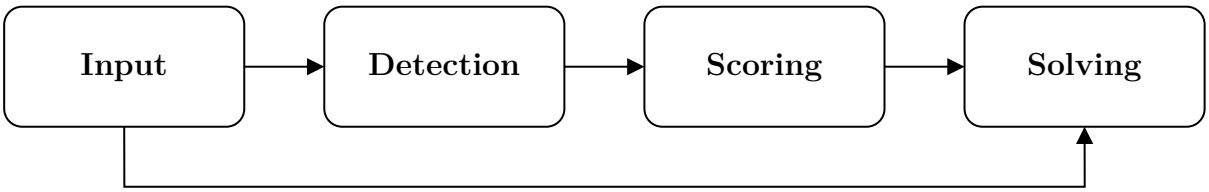


Figure 4.1: A simplified overview of the four major stages of solving a model with GCG.

In this chapter, we introduce Generic Column Generation (GCG), a decomposition solver which is based on the open-source MIP-Solver SCIP (SCIP) [2]. 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) [3] or commercial software such as *SAS* [4] 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) [5].
- 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 [6].

Kurz die 4 Schritte aus Bild erwähnen und einen Satz

4.1 Detection

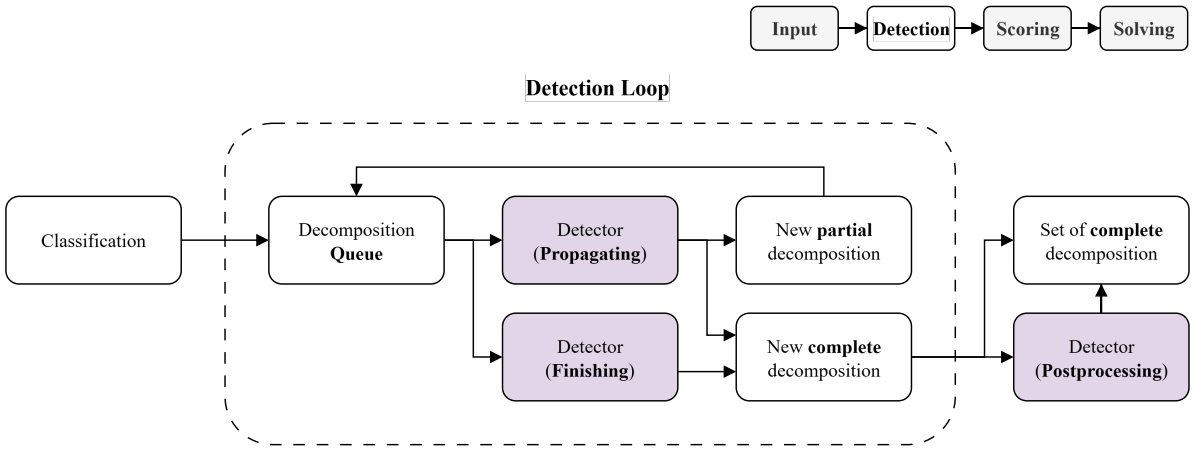


Figure 4.2: A simplified overview of the detection process and its detection loop.

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 [6]. 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.

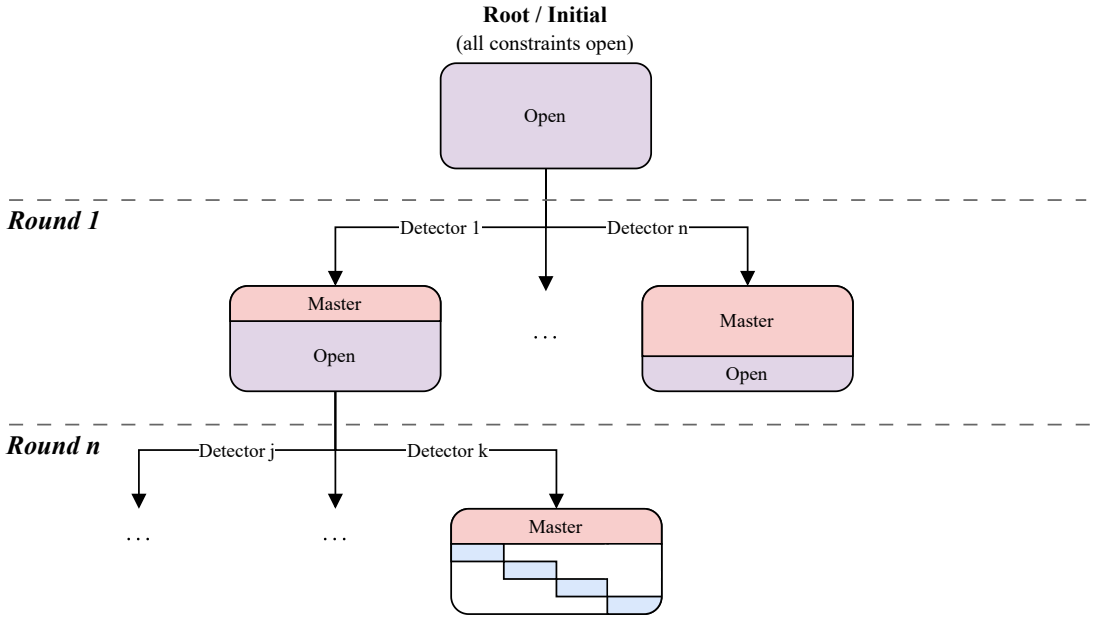


Figure 4.3: Visualization of the induced tree of propagated partial decompositions.

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 [6].

Grammatik Wort-wahl

4.2 Classifiers

4.2.1 Name Classifiers

4.2.2 Numeric Classifiers

In contrast to a classifier based on properties the lassifiers are solely based on

4.2.3 Type Classifiers

Classifiers based

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 the *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* [2]. 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 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 [7].

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

[2] 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.

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 order to get a more fine-grained classification based on inferred constraint types,

4.2.4 Detectors

4.3 Existing Detectors

4.4 Example

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

Figure 4.4: 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.

In order to illustrate the detection

5 Implementation

5.1 Architecture

A

A

5.2 Classifiers

A

A

5.3 Pre-Processing

5.4 Tree Refinement

A

A

A

5.5 Post-Processing

5.6 Scoring

5.6.1 Entropy Score

5.6.2 Connected Block Score

6 Evaluation

6.1 Setup

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

Table 6.1: Consumer-grade components used to run all experiments.

All experiences were run on a system with components as specified in Table 6.1.

6.2 StrIPlib

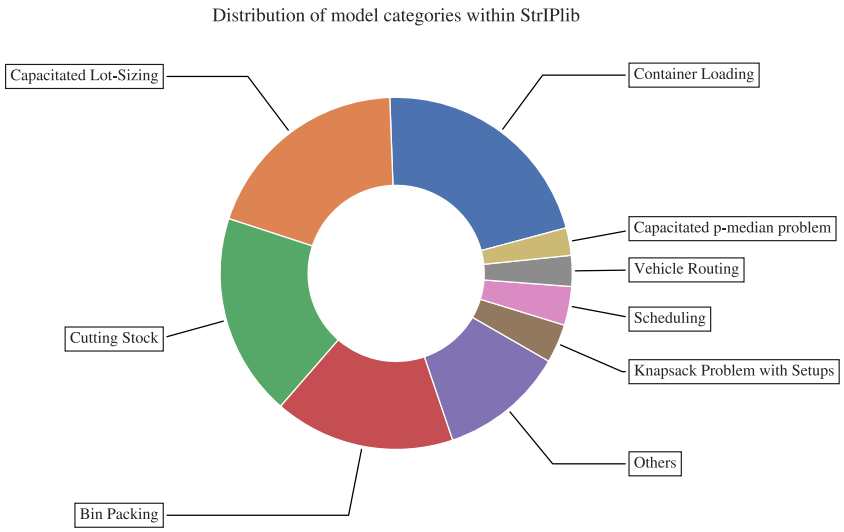


Figure 6.1: The distribution of different categories of model within strIPlib. Most problems are part of "common" categories like Bin-Packing, Scheduling and Routing. The category "Others" includes e.g. Fantasy Football, Train Scheduling and different types for which only a small number of model files 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 6.1. 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 (Problem-

6.3 Stuff

7 Notes

7.1 MIPLIB Constraint Types

7.2 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 7.1: A relaxed version of the constraint types MIPLIB uses.

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

[2] "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).

[3] 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.

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

[5] 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).

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

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