Titel

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:

Titel

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 November 13, 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
SRT	Set Refinement Tree

List of Figures

List of Tables

Quellcodeverzeichnis

1 Introduction

Feel introduced.

1.1 Motivation and Contribution

I am motivated.

1.2 Structure

This thesis is organized into seven main chapters. Chapter 1 introduces the motivation, contributions, and overall objectives of the work. Chapter 2 provides the theoretical foundations necessary for understanding the proposed methods, including concepts from linear optimization, graph theory, hashing, and algorithmic parallelization.

Chapter 3 reviews related research and situates this work within the broader scientific context. Chapter 4 presents the framework of GCG, discussing its underlying principles, detection mechanisms, and illustrative examples. Here, the primary focus is a sufficient understanding of the algorithmic principles and the current state *as is*.

Building on this, Chapter 5 describes our Tree Refinement approach in detail, outlining its algorithmic design, classification strategies, and evaluation criteria. We build upon the principles discussed in Chapter 4 to design an algorithm which can be seamlessly integrated into the existing framework. Chapter 6 focuses on implementation aspects, covering system architecture, data structures, concurrency mechanisms, and methods for ensuring computational efficiency and avoiding unnecessary computations.

Finally, Chapter 7 evaluates the proposed methods through experimental studies, detailing the setup, datasets, and performance analysis using instances from the StrIPlib and MIPLIB libraries. The thesis concludes with a summary of findings and potential directions for future research.

1.3 Hm

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

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

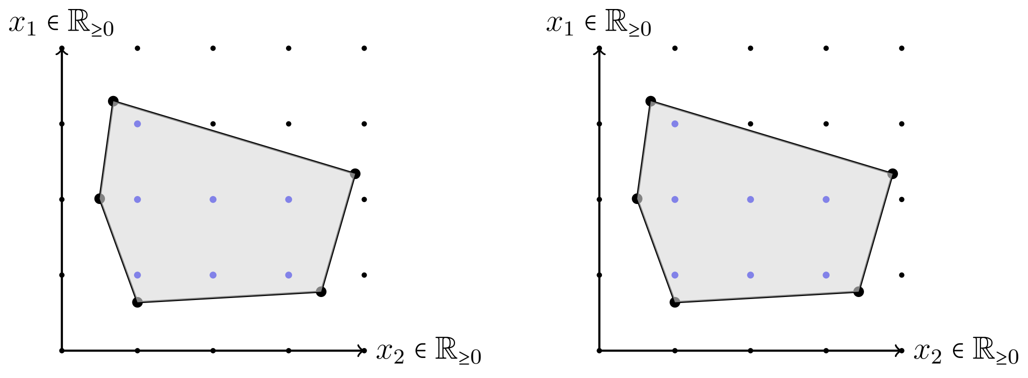
Linear Optimization is a mathematical optimization technique used to determine the best possible values for a set of variables in a given model, whose constraints or requirements are represented by linear relationships. The goal is typically to maximize or minimize a objective function, subject to a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

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

Even thought linear programs are already a powerful tool on its own, some problems require *discrete* bounds for some variables, e.g., the y-variables shown in the system above. These variables are usually restricted to a subset of $\mathbb{Z}$. If the system contains integer *and* continuous variables, then the model is called a *Mixed Integer Program* (MIP). If only discrete variables are present, the prefix "Mixed" is omitted.

MIPs are significantly more difficult to solve than pure linear programs, since the integer restrictions make the feasible region *non-convex*, eliminating a key assumption of the simplex algorithm and its optimality conditions. Standard solution approaches include branch-and-bound, branch-and-cut, and cutting-plane methods, which systematically explore and prune the search space.

It can be shown that the decision problem of whether an **IP!** with all variables restricted to the domain $\{0, 1\}$ has a solution is NP-hard. Despite the higher computational complexity, MIPs are extremely powerful because they allow the modeling of a wide range of practical decision-making problems, including scheduling, routing, facility location and production planning

2.2 Dantzig-Wolfe Decomposition

The *Dantzig-Wolfe Decomposition* is a thing

Seite 2

Seite 3

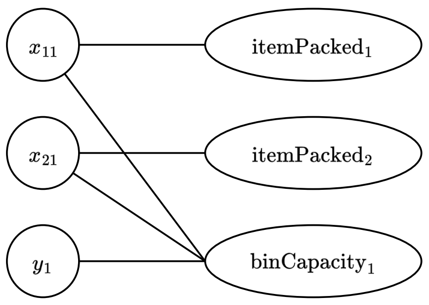
2.3 Graph Theory

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 $V \subseteq \{1, 2, \ldots, n\}$ 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 constraint matrix $A \in \mathbb{Q}^{n \times m}$ can be encoded as a bi-partite graph $\mathcal{G}_A = (V, E)$ with $E = \{\{var, cons\} \mid var \in \text{rows } A, cons \in \text{cols } A\}$ and one node for each variable/constraint as shown in Figure 2.2.



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

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

2.4 Hashing

Similar to graph data structures, hashing is a fundamental technique in computer science that transforms data of arbitrary size into a fixed-size sequence of bits of length k, called a *hash value*. This value is computed by a so called *hash function*, in the following denoted $h : X \to \{0,1\}^k$, which maps elements $x \in X$ to a fixed-sized bit-string. The primary purpose of hashing is to enable efficient *data retrieval*, verification, and comparison without requiring direct access to the original data. Data retrieval in particular is of vital interest in Chapter 6.

For the purpose of this work, a "good" hash function must meet the following requirements:

- *Determinism*: The same input always produces the same output.
- *Efficiency*: The function should compute the hash quickly, even for large inputs.
- *Collision resistance*: Due to the image of h being of fixed-sized, collisions, i.e., two different inputs resulting in the same hash value, are usually unavoidable. A good hash function should minimize the likelihood of this event.

Hashing is widely applied in multiple domains. In data structures, hash tables use hash functions to achieve average-case constant-time complexity for insertion, deletion, and search operations. Furthermore, hashing is a central part of domains like cryptography, finance and authentication systems. cite

In practice, the input domain X must not need to be arbitrary. Instead, domains like 32-bit unsigned integer or a sequence of characters (e.g. strings) are often sufficient. For this work, we will need a hash function for the former case which full-fills the requirements mentioned above. A possible implementation of a simple hashing function using only multiplications and xor-shifts is shown in Algorithm 1. The magic numbers `0x7feb352d` and `0x846ca68b` provide a low *avalanche score* in practice [2], i.e., the number of bits that *do not* change if one bit in the input is flipped [3].

In case a hash for more complicated structures like lists or arrays is required, Algorithm 2 can be used instead. It is a combination of Algorithm 1 and the frequently used `boost::hash_combine` function from the Boost C++ Framework [4].

Algorithm 1 A function to hash 32-bit unsigned integers.

Input: Unsigned 32-bit integer x.

Output: Hash value of x, which is an unsigned 32-bit integer as well.

```
function HashSingle($x$)
    $x \leftarrow (x \oplus \textsc{ShiftRight}(x, 16)) \cdot \text{0x7feb352d}$
    $x \leftarrow (x \oplus \textsc{ShiftRight}(x, 15)) \cdot \text{0x846ca68b}$
    $x \leftarrow (x \oplus \textsc{ShiftRight}(x, 16))$
    return $x$
end function
```

Algorithm 2 A function to combine the hash values of multiple objects [4].

Input: List of objects L, hash function $h : L \to \mathbb{N}$.

Output: Combined hash $x \in \mathbb{N}$ of objects in L.

```
function HashList($L$)
    hashValue ← SizeOf($L$)
    for $x \in L$ do
        current ← HashSingle($x$)
        left ← ShiftLeft(hashValue, 6)
        right ← ShiftRight(hashValue, 2)
        hashValue ← $x$ + 0x9e3779b9 + left + right
    end for
    return hashValue
end function
```

2.5 Parallelization of Algorithms

Parallelization is a concept frequently used on modern computer systems to make an algorithm more efficient in terms of its total execution time. By splitting up the work usually done by a sequential algorithm running on a singular core into multiple, often independent sub tasks, one can execute these tasks at the same time on more than one core.

Algorithm 3 A simple example showing how writing to a shared variable can lead to a *race condition.*

```
function COUNTUP(sharedCounter, n)
    $sharedCounter \leftarrow sharedCounter + 1$
end function
```

If the access is not coordinated and multiple thread write to shared data at the same time, the final result can become unpredictable as shown in Algorithm 3. In this example, our goal might be to increment an integer to 100 as fast as possible in increments of one. Here, we create 10 threads in total, each executing CountUp(10). The increment done in the algorithm usually consists of three *separate* operations:

1. Read the value of *sharedCounter*.
2. Increment the value by 1.
3. Write the result back to *sharedCounter*.

Assuming a modern operating system, a thread might get just enough execution time to execute operation 1, before it is re-scheduled at a later point in time, while other threads continue to increment the counter. If the thread gets finally scheduled again, it increments the now outdated value by 1 and writes it back, undoing all the progress. This exact situation is usually called a *race condition*, i.e., when multiple threads are accessing shared data and the final result depends on the timing or order of their execution.

Mutex

A *mutex* (short for mutual exclusion lock) is an important synchronization primitive used to enforce exclusive access to shared data, which is usually required if multiple threads operate on the same data structure in parallel. An implementation of a mutex usually provides two essential operations:

1. `lock()`, to acquire the mutex and gain exclusive access. The operation must either fail or block the calling thread If it was already acquired by a different thread.
2. If the acquiring thread has executed all operations related to the shared data, it must release its lock via. `unlock()`. A failure to do so might lead to state called a *deadlock*, a state in which no further progress is made because all threads are waiting to acquire a lock.

Efficient use of mutexes requires minimizing the time spent within critical sections, thereby reducing contention and improving overall parallel performance.

Condition Variable

While mutexes handle mutual exclusion, condition variables enable threads to coordinate based on the occurrence of certain events or conditions, usually called *predicates.* A condition variable is always associated with a particular condition or predicate, which allows a thread to suspend execution until the condition becomes true, or to notify waiting threads if the status of the condition changed, e.g., because it itself modified the state some shared data. Condition Variables are always used in conjunction with a mutex and provide two essentials operations:

1. `wait()`, which acquires the mutex and checks the predicate. If it evaluates to true, we continue the execution. If it evaluates to false, the thread is suspended [1].
2. `notify()`, which signals to waiting/suspended threads that the shared data changed and the predicate might evaluate differently now. Each awaken thread must acquire the mutex again and re-evaluate the condition.

[1]Effects such as *spurious wake-ups* must be handled as well, but the details are not of importance for this work.

2.6 Partition Refinement

Partition refinement is a fundamental concept in computer science, particularly relevant in fields such as automata theory [5], graph theory, and model checking [6]. A *partition* refers to a decomposition of a finite set U into disjoint, non-empty subsets $\{A_1, A_2, \dots, A_k\}$, 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 $\pi = \{A_1, A_2, \dots, A_k\}$ of a set U is called a refinement of another partition $\pi' = \{B_1, B_2, \dots, B_m\}$, 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 $f : P \times Q \to \Pi(A)$ 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 $\pi_f = \{A_1, A_2, \dots, A_k\}$ 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 $\mathit{Step} : \Pi(U) \to \Pi(U)$ 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 4 A simple partition refinement algorithm which refines π_{init} until a fixed-point is reached.

Input: Initial partition $\Pi_{init} = \{A_1, A_2, \dots, A_k\}$, splitter-function $f : P \times Q \to \Pi(P)$
Output: 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 4. Note that new cells are continuously being produced in the loop which are able to act as inducing cells during the next iteration.

Re-work Pseu-docode

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 [7] which also contains corresponding correctness and termination proofs. For this case in particular, Algorithm 4 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

roter faden

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 [6] and sorting algorithms [8].

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 hinted on in [9]. 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 5.

Algorithm 5 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
```

2.7 Surprise and Entropy

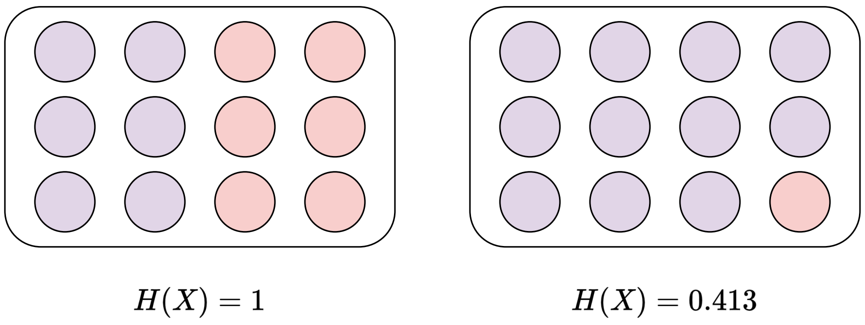


Figure 2.3: Entropy is measure of "surprise" and it increases with decreasing probability. On the left side, both colors are evenly distributed, so drawing either one is equally surprising. On the right side, drawing a red ball from the set of elements would be very surprising, because the probability 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 [10].

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

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

2.8 Adjusted Rand Index

The *Rand Index* is a statistical measure used to compare two different partitions $\pi = \{A_1, A_2, \ldots, A_k\}, \pi' = \{B_1, B_2, \ldots, B_l\}$ of elements from the same set $U = \{1, 2, \ldots, n\}$. Let $f_\pi(x) : U \to \mathbb{N}$ 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:

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

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 [11].

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

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 6.

$\pi' \setminus \pi$	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 6 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 $f_C : U \to \mathbb{N}$ 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 [12] for a more detailed discussion on different similarity measures and their mathematical derivation.

3 Related Work

4 Generic Column Generation (GCG)

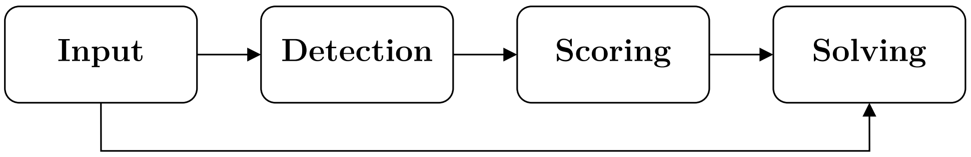


Figure 4.1: A simplified overview of the four major stages of solving a model with GCG.

In this chapter, we introduce Generic Column Generation (GCG), a decomposition solver which is based on the open-source MIP-Solver SCIP (SCIP) [13]. 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) [14] or commercial software such as *SAS* [15] 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) [16].
- 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 [17].

Kurz die 4 Schritte aus Bild erwähnen und einen Satz

4.1 Detection

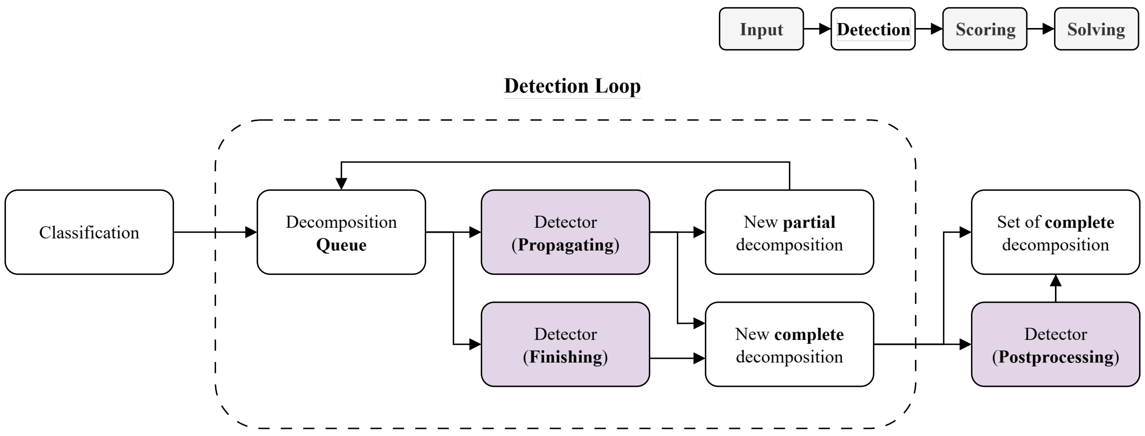


Figure 4.2: A simplified overview of the detection process and its detection loop.

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 [17]. 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.

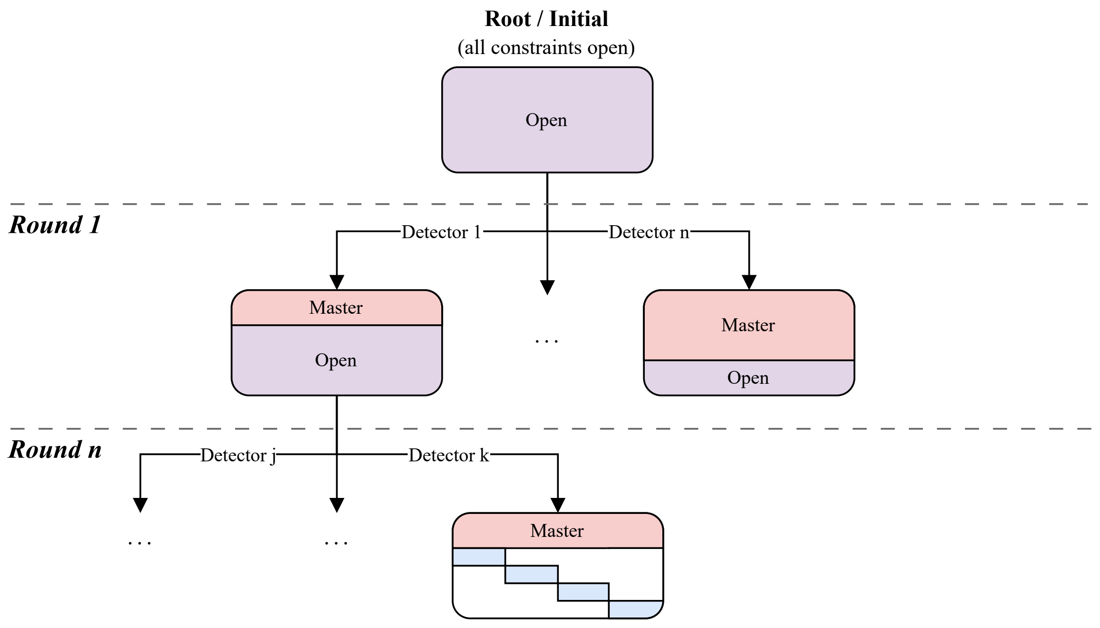


Figure 4.3: Visualization of the induced tree of propagated partial decompositions.

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 [17].

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.

Furthermore, each classifier is identified with an unique *name* and an integral priority, influencing the order in which the classifiers are being executed by the framework.

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.

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

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 . If the modelers provided no names of it's own, then the underlying solver usually chooses a default prefix such as "c" or "cons" for constraints, followed by an increasing number, resulting in constraint names "c0" - "c4999" for a model with 5000 constraints.

Appendix

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

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

Equation 4.1 can be computed in $O(|w| \cdot |v|)$ using a dynamic programming approach. Let $B = (\operatorname{lev}(\operatorname{name}(c_i), \operatorname{name}(c_j)))_{1 \le i,j \le m}$ the pair-wise Levenshtein Distance between constraint names, $k \in \mathbb{N}$ the *connectivity* and $G = (V, E)$ with $V = \{c_1, c_2, \ldots, c_m\}, E = \{\{u, v\} \mid u, v \in V, u \neq v, \operatorname{lev}(\operatorname{name}(u), \operatorname{name}(v)) \le k\}$. 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 $s = v$:

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

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.

WIP

5000 cons limit

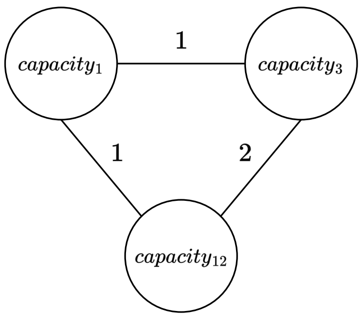


Figure 4.4: The graph of pair-wise Levenshtein weights for three capacity constraints. For $k = 1$, the edge between $capacity_3$ and $capacity_{12}$ vanishes, but because they is still a connecting path via. $capacity_1$, both constraints are assigned to the same class.

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

Example

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

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. Example

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 [18]. 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

[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.

MIPLIB Constraint Types

Nr.	Type	Linear Constraint	Notes
1	Empty	$\emptyset$	-
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 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 [19]. 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, even thought it does not seem like it on a surface level. 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.

Explain issue of GCG fixed zero variables

4.3 Existing Detectors

Building upon the different kinds of classifiers mentioned in the previous section, we continue with a brief summary of existing detectors that have been developed and integrated into the framework since its creation. Each detector follows a distinct idea of how structural information can be identified and exploited, providing complementary perspectives on the decomposition of mixed-integer programs. We do not provide a full list of all available detectors, but only focus on the most relevant for this work.

4.3.1 Power Set Detectors

Title

The term *Power Set Detectors*, is a term only used in this section and is not used in any official documentation, but it emphasizes the core of their behavior. They operate by exploiting pre-computed classifications of constraints and then generating partial decompositions that assign some constraints or variables to the master problem and leave others open, i.e., in general, they only output partial decompositions.

In more detail: it cycles through each classifier and, for *each subset* of constraint classes it creates a decomposition in which all constraints of those classes are fixed to the master. In addition to that, classifiers may mark constraints or variables as "master only", which are also assigned to the master problem. All remaining constraints not in the current subset remain unassigned (open). Thus it explores combinations of "which constraint classes go into master" vs "which go into blocks".

By doing so, it aims to identify decompositions where groups of constraints move coherently together (because they share class membership) and thus suggest a meaningful split into master and block parts. In aggregate, the detector provides heuristic suggestions for decompositions by leveraging the class structure of constraints rather than purely graph-cut or adjacency heuristics. It does not guarantee that the decomposition is complete or optimal; rather, it produces candidates of partial decompositions which then may be refined by other detectors.

An example involving such a detector is explained more in-depth in Section 4.4.

4.3.2 Connected Components Detector

The *Connected Base* detector focuses on completion by breadth-first search (BFS) in the dependency graph of constraints and variables, which was already shown in Figure 2.2. It takes as input a *partial decomposition* and tries to complete the assignment by selecting all open constraints and variables reachable from already assigned elements (via adjacency) and assigning them accordingly.

Functionally, it walks the adjacency frontier: given an *open* constraint c_i, it computes the set of all reachable constraints $\text{reach}(c_i)$. A constraint c_j is considered reachable from c_i iff they either share a common variable *or* if there is some constraint c_k which reachable from both c_i and c_j. This process is iterated until all such sets $R = \{\text{reach}(c) \mid c \in Cons\}$ are found and a new block is created for each $r \in R$. Afterwards, all remaining open variables are assigned to either the first block (it it exists), or to the master.

4.3.3 Hypergraph METIS (hMETIS) Detector

The hypergraph-partitioning detector (using the external tool **hMetis!** (**hMetis!**)) treats the problem's constraint–variable incidence structure as a hypergraph: constraints and variables become nodes, and hyperedges represent the incidence relations. Then it applies a heuristic partitioning of this hypergraph into multiple components (blocks) aiming to minimise the coupling (i.e., hyperedges) between blocks. Each block becomes a candidate for a subproblem (pricing) and the remainder becomes the master. Thus the detector identifies "natural" substructures in the problem where variables/constraints are more tightly connected internally than externally. In practice, this leads to decompositions where the incidence structure suggests a separation into loosely-coupled modules. Because the hypergraph partitioning is heuristic, the resulting decomposition is approximate and depends on the connectivity pattern of the original MIP matrix. But for highly modular problems it can yield good block structures.

4.4 Example

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

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

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.

Wording

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, GCG will transition to the detection stage. Here, the *cons class* detector will yield 3 new partial decompositions as shown in Table 4.2, 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. This process is illustrated in Figure 4.7.

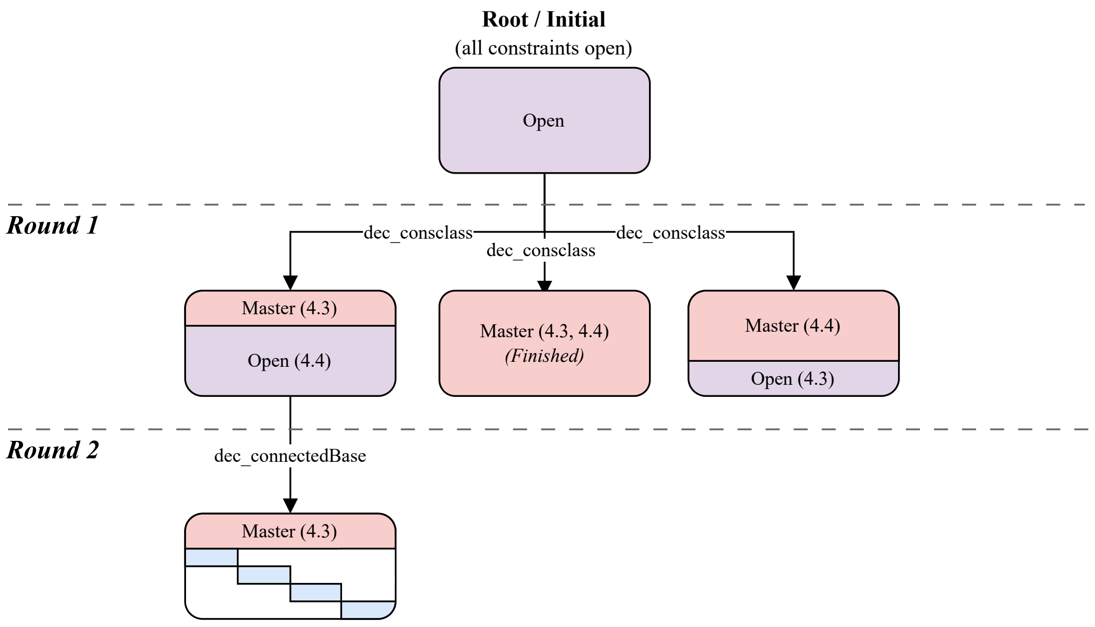


Figure 4.7: A possible tree of partial decompositions for a textbook Bin-Packing model.

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. This can be seen as a generalization of the approach presented in [9]
- 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 The Algorithm

In [9], the implemented approaches can be seen as a three-step process:

1. Transform a given model to a graph-based representation.
2. Select a suitable initial partitioning.
3. Choose *one* splitter-function and run the standard partition refinement algorithm until a stable partition is reached.

Depending on the type of model, running the refinement with different initial partitions and splitter-functions might be necessary, which was already recognized in [9]. Furthermore, we suspect that for some models it might even be required to use different splitter-funtions on different *parts* of the model.

In the following, we will generalize this process by adopting a similar approach as the one shown in Chapter 4. Instead of choosing *one* way of refining the constraints or variables based on an initial partition, we try different *strategies* to explore a more broader search space. Afterwards, the found partitions are scored using a family of scoring functions $g_i : \Pi(U) \to \mathbb{R}$ and the most promising ones are selected. In addition to a scoring function, each score must provide an individual *total order* on $\mathbb{R}$, to support scores for which higher values reflect better partitions and vice versa. The ordering ensures a well-defined meaning of "promising" for each score. Note that the overall goal remains unchanged: Based on an initial partition, we aim to iteratively refine the cells in such a way, that ultimately the constraints or variables in each cell belong to the same semantic grouping as the modeler intended to.

In order to achieve this, we propose multiple *strategies* which form the basic building blocks of the algorithm and are discussed in Section 5.4. Each strategy is defined by a function $f_{strat} : C_i \to \Pi(C_i)$ taking a single cell C_i as input and computes a partition $\pi_{C_i} \in \Pi(C_i)$ as output, thereby refining the cell. In the following, the process of refining a cell according to a certain strategy is sometimes referred to as "expanding a cell". The partitions of the cells are organized in a Set Refinement Tree (SRT) as shown in Figure 5.1, i.e., each node, with exception of the root node, corresponds to a possible partitioning of its parent cell, which was computed by a strategy. The root node and its cells correspond to the initial partition and it is the *only* node for which the union of its cells corresponds to the whole set of constraints or variables. The cells of all other nodes only partition its immediate parent cell in the tree. This process implies that an additional post-processing step is required which translates the tree of cell-refinements to actual partitions. This step is discussed in Section 5.6.

In order to use a unified notation in the following sections, we will define the SRT and its associated information more formally. A SRT can be formalized as a tuple $T = (V, E, U, R, S)$ with universe U corresponding to the set of all constraints or variables, designated root node $R \in V$ and set of strategies $S \subseteq \mathbb{N}$. The tuple (V, E) must induce an directed acyclic graph. Each node $v \in V \setminus R$ of the tree is associated with a parent cell $\text{ParentCell}(v) \in 2^U$, parent node $\text{ParentNode}(v)$, a set of cells $\text{Cells}(v) \in \Pi(\text{ParentCell}(v))$ and the strategy $\text{UsedStrategy}(v)$ that was used to compute v from its parent. For the root node we require $\text{Cells}(R) \in \Pi(U)$, i.e., a valid partition of U must be used to initialize the tree.

Algorithm 7 A high-level overview of the algorithm. All additional data structures, optimizations and handling of necessary metadata was omitted.

Input: Initial partition π_{init}, set of strategies $S = \{f_1, f_2, \ldots, f_k\}$ with $f_i : P \mapsto \Pi(P)$
Output: List of partition Π

```
function TreeRefinement($\pi_{\text{init}}$)
    Init empty SRT $T = (V, E, U, R, S)$
    $queue \leftarrow$ Create queue with element $\pi_{\text{init}}$
    while Size($queue$) $\geq 0$ do
        $\pi_{\text{current}} \leftarrow$ Pop($queue$)
        for $cell \in \pi_{\text{current}}$ do
            for $f_{\text{strategy}} \in S$ do
                $refined \leftarrow f_{\text{strategy}}(\pi_{\text{current}})$        ▷ Section 5.4
                Push(queue, refined)
            end for
        end for
    end while
end function
```

WIP

wrong image

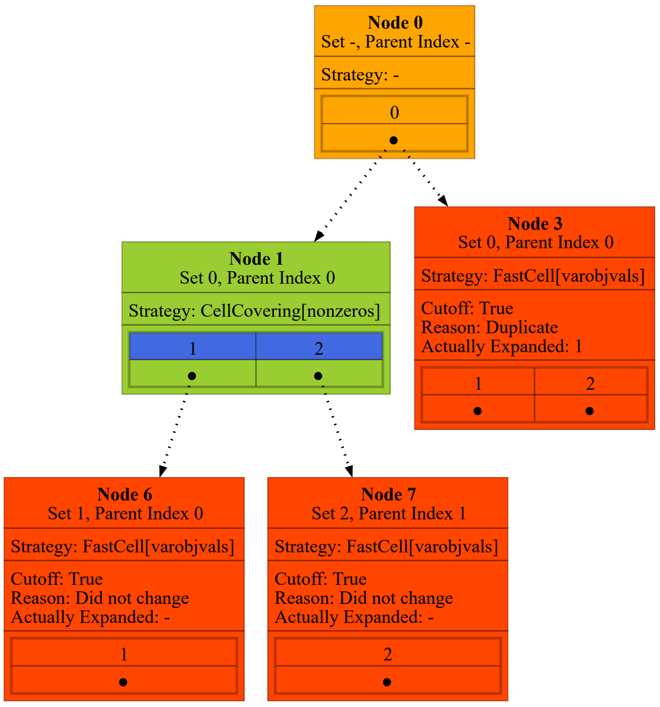


Figure 5.1: WIP WIP WIP A example of a simple refinement tree for the Bin-Packing Problem.

5.2 Information

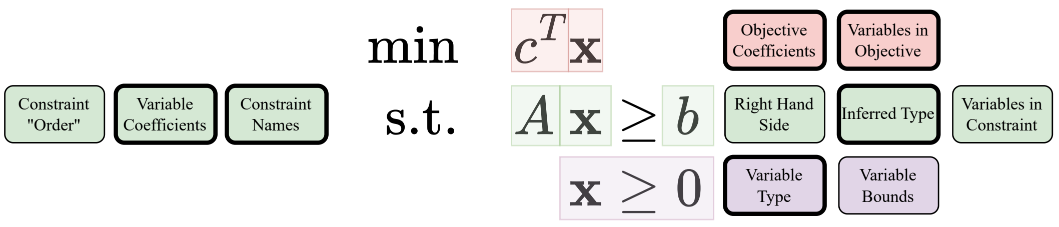


Figure 5.2: All parts of a model that contain useful information for semantic grouping of constraints and variables. Elements with a thick border are already used as a key concept in one of the existing detectors.

Before we present any algorithmic details, we give an overview about the available information which might be used to define suitable strategies.

roter Faden

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.

5.3 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. Adding new classifiers has, in addition to practical implications such as higher maintenance overhead, additional side-effects regarding runtime and memory requirements. Each new classifier provides new opportunities for existing detectors to find new partial decompositions. This can prove especially useful for detectors like consclass, which directly depend on the found classifications. On the other hand, this dependence can result in a sub-optimal runtime investment if the found classifications provide no new information, because the generated partial decompositions based on that will most certainly be of bad quality as well. Therefore, we will refrain from mentioning all missing model properties for which a classifier *could* be built, and only focus on promising candidates.

wording

Summarize goals for custom classifiers

5.3.1 Bounds

When considering bounds, we differentiate between variables and constraint respectively.

Variable bounds

The classifier for variable bounds can be considered as a simple mapping from pairs of bounds to a unique class index. More formally, given a list of all unique pairs of lower and upper bounds for variables $B = (b_1, b_2, \ldots, b_k)$, with $b_i \in \mathbb{Q} \times \mathbb{Q}$, we map each variable to the index of its bounds in B. This mapping is trivially unique.

Mapping the variables in the described way has the side-effect that $0 \leq x \leq 1$ and $y \in \{0, 1\}$ are mapped to the same class. This behavior is able to account for missing declarations of variables as binary in the input format read by GCG. The classification is wrong if x is truly meant to be a continuous variable, but this is offset by the fact that GCG already includes a classifiers based on SCIP types, which will correctly classify both variables in this case.

Types of models where information about the variable bounds *can* be leveraged include instances of e.g. Container Loading. Here, variables $x, y, z \in \mathbb{R}$ encoding the positions of parcels to be loaded into a container are continuous. If the container is not a cube, i.e., it has a different length in each spacial dimension, then the variables have to have different bounds.

Constraint bounds

A classifier for constraint bounds works in a similar manner as for variables. By collecting all bounds and assigning a class to each constraint based on that list, we obtain a unique mapping. Note that for inequalities the absolute value either of the two bounds will always be infinity. For equalities which were not replaced with equivalent inequalities, both bounds will be equal.

In addition to a classifier based on the actual *values* of the bounds, i.e., for values $a, b \in \mathbb{R}$ of a linear constraint $a \leq \sum a_i x_i \leq b$, we propose a classifier based on the *sign* of a and b. Here, the linear constraint is transformed to standard form to prevent a different classification for equivalent constraints in case the constraint is multiplied by -1. Therefore, only four potential classes shown in Table 5.1 remain. This classifier can be used for variables analogously.

Class Nr.	**Name**	$sign(a)$	$sign(b)$
1	Positive	+	+
2	Mixed	+	−
2	Mixed	−	+
3	Negative	−	−
4	Zero ($a = b = 0$)	+/−	+/−

Table 5.1: The four classes of the sign-variant of the bounds classifier.

cap. lot sizing example

5.3.2 Relaxed MIPLIB types

Nr.	Type	Linear Constraint	Notes
1	Empty	$\emptyset$	-
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	$\sum 1x_i \leq b$	$\forall i : x_i \in \{0, 1\}, b \in \mathbb{N}_{\geq 2}$

Table 5.2: The structure of all constraint types which the *relaxed* version of the MIPLIB classifier detects.

table

As outlined in Section 4.2, the classification of constraints and variables plays a key role in structure detection of mixed-integer programs. The MIPLIB types provides a detailed list of constraint types, ranging from simple set-packing relations to complex mixed-binary expressions. This fine-grained categorization can be counterproductive when used directly within a detection framework, because excessive granularity can lead to fragmentation of the constraint set: two constraints that are functionally similar but formally classified as different types would end up in separate partitions, preventing the discovery of broader structural *patterns*. To address this, we propose a simple but functional similar classifier which *deliberately* defines a smaller and more permissive set of constraint types as shown in Table 5.2. Each type is allowed to encompass a wider range of constraint types, i.e., we "allow" more deviation from the original MIPLIB types. In other words, the classifier merges several fine-grained MIPLIB categories into coarser types, without loosing to much information. Similar to the MIPLIB classifier, all types can be deduced from local information only. Thus, one pass over the constraint matrix is sufficient to classify all constraints.

5.3.3 Voting

As mentioned earlier, the motivating idea behind each individual classifier is to group constraints according to some property, while the type of this property can vary greatly between classifiers. Under the assumptions that groups of constraints share at least *some* of these properties and are thus assigned to the same classes, we can define a new type of classifier. Note that this classifier can be equivalently conceptualized as an additional *strategy*, which are the basic building blocks of the refinement algorithm and explained in Section 5.4. This assumption is, at least on examples that can be observed in practice, a reasonable one.

Voting (unordered)

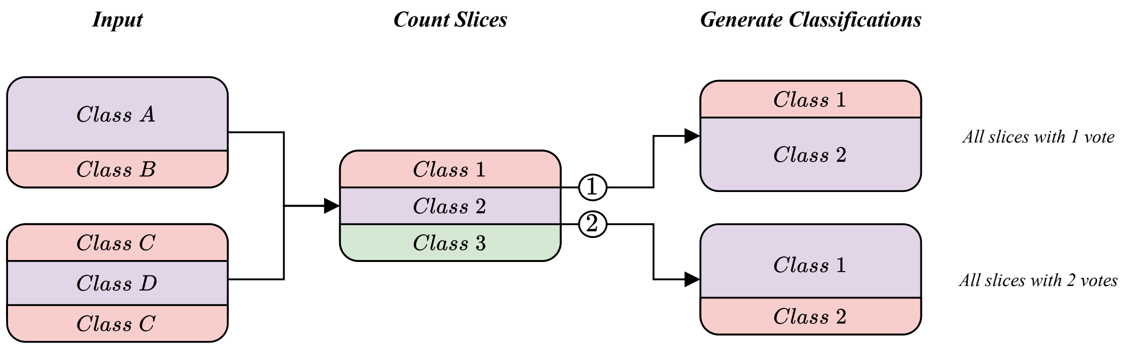


Figure 5.3: text

In order to derive a simple and efficient algorithmic approach to this problem, we can make an additional assumptions not based on the model itself, but on its representation in common input formats like `.lp` and `.mps` files. A lot of models are generated via. scripts and written to such files for portability and interoperability with other solvers. One exploitable property which can be observed in a lot of models is, that groups constraints are usually *added in bulk*. This results in consecutive blocks of constraints which are semantically related, which is illustrated in the “Input” column of Figure 5.3. If a class of constraints is split into two non-consecutive groups like the second partition from Figure 5.3, then we can deduce that they are meant to be two different groups.

Voting (unordered)

If the read model does *not* have the property of properly ordered constraint blocks, a more involved algorithmic approach can be used. Here, the general problem can be reduced to group constraints with each other which are *often* assigned to the same class. Given a set of partitions $Q = \{\pi_1, \pi_2, \ldots, \pi_k\}$, we can compute a matrix $A = (n_{ij})_{1 \le i,j \le m}$ which contains the pair-wise occurrences of all elements with m being the number of constraints in total. Here, the set of partitions Q might correspond to the found classifications in e.g. GCGs classification stage. Each entry n_{ij} can be computed by using Equation 5.1.

$$n_{ij} = |\{x \mid \forall \pi \in Q, i \in \pi, \forall x \in \pi\}| \tag{5.1}$$

If all entries are computed the matrix requires $O(m^2)$ space in memory and $O(m^2)$ runtime, which is not feasible for large models. A simple method to reduce the memory requirements of A is the concept of a *sparse matrix*, i.e., only those $n_{ij} \neq 0$ are actually held in memory. Furthermore, computing the all entries n_{ij} results in lot of unnecessary computations, as we are only interested in $\max\{n_{i1}, n_{i2}, \ldots, n_{i,i-1}, n_{i,i+1}, \ldots, n_{i,m}\}$ for each constraint $1 \le i \le m$.

For sets of partitions in which the individual cells do not "agree" with each other, i.e., it holds that $n_{ij} \ge 1$ for almost all pairs, we propose a different approach with $O(m)$ space complexity but slightly worse runtime complexity of $O(m^2|Q|)$. In practice, however, the approach is expected to be much faster. We assume the existence of a fast lookup table $f_\pi : \{1, 2, \ldots, m\} \to C_i$ which maps a constraint or variable to its cell in π. Instead of computing each n_{ij}, we propose to construct each row individually as shown in Algorithm 8. The algorithm looks though all constraints $1 \le i \le m$ and finds *a* constraint $1 \le j \le m, i \neq j$ which is most often part of the same cells as i. Afterwards, a union-find like procedure can be applied to successively merge constraints with their top partner.

why

cite union find

Algorithm 8 Algorithm for constructing a partition in which two constraints c_i, c_j are in same cell if $n_{ij} = \max\{n_{i1}, n_{i2}, \ldots, n_{i,i-1}, n_{i,i+1}\}$ and vice. versa.

Input: Set of partitions $Q = \{\pi_1, \pi_2, \ldots, \pi_k\}$

```
function VotingUnordered(Q)
    $mostFrequentPartners[1, \ldots, m] \leftarrow m$ Integers
    for $x \in \{1, 2, \ldots, m\}$ do
        $occurences \leftarrow m$ Integers
        for $\pi \in Q$ do
            $cell \leftarrow f_\pi(x)$
            for $y \in cell$ do
                if x = y then
                    continue
                end if
                $occurences[y] \leftarrow occurences[y] + 1$
            end for
        end for
        $mostFrequentPartners[x] \leftarrow \max_{i \in \{1,2,\ldots,m\}} occurences[i]$
    end for
    Merge constraints based on $mostFrequentPartners$ with union-find approach
    return Partition after Union-Find procedure
end function
```

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 can be formalized as a function $f : S \to \Pi(S)$ which gets a single set $S \subseteq U$ as input and produces a partition $\pi \in \Pi(S)$ as output. Conceptually, each strategy can be seen as a materialization of a specific splitter function as defined in Section 2.6.

5.4.1 Slice (Partition)

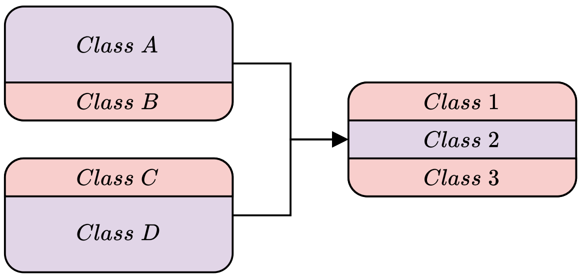


Figure 5.4: A simplified illustration assuming that constraint in both partitions can be rearranged into continuous blocks, thus “slicing” the partitions into different-sized chunks. The output partition inherits all these slices.

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:

$$\pi \sqcap \pi' = \{A_i \cap B_j \mid A_i \in \pi, B_j \in \pi'\} \setminus \varnothing \tag{5.2}$$

This concept is illustrated in Figure 5.4. Because strategies only refine single sets and not whole partitions, Equation 5.2 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 5.3 and computed efficiently using Algorithm 9.

$$f_{\pi_{splitter}}(S) = \{A_i \cap S \mid A_i \in \pi_{splitter}\} \setminus \varnothing \tag{5.3}$$

Note that the operator $\sqcap$ is trivially associative, i.e., $(\pi \sqcap \pi') \sqcap \pi'' = \pi \sqcap (\pi' \sqcap \pi'')$:

$$\begin{aligned} X \in (\pi \sqcap \pi') \sqcap \pi'' &\iff \exists Z \in (\pi \sqcap \pi'), C \in \pi'' : X = Z \cap C \\ &\iff \exists A \in \pi, B \in \pi', C \in \pi'' : X = (A \cap B) \cap C \\ &\iff \exists A \in \pi, B \in \pi', C \in \pi'' : X = A \cap (B \cap C) \\ &\iff X \in \pi \sqcap (\pi' \sqcap \pi'') \end{aligned}$$

In conjunction with commutativity, this fact can be used to e.g. eliminate part of the search space by pruning duplicated nodes in the tree and only expanding one instance of any given sub-tree.

A simple example where two successive applications of this slicing strategy can

Algorithm 9 If a lookup table represented by function f is available, then Function 5.3 can be implemented in $O(|S|)$.

Input: Partition $\pi = \{A_1, A_2, \dots, A_k\}$, set $S \subseteq U$, 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: Partition of S according to Function 5.3.

```
function STRATEGYSLICE($\pi, S$)
    $\pi_{out} \leftarrow$ list of $k$ empty sets $B_1, B_2, \dots, B_k$
    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)

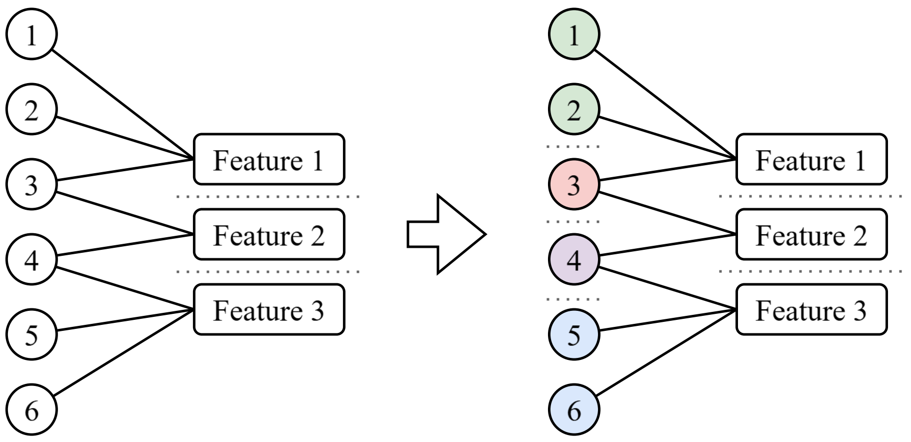


Figure 5.5: Given $U = \{1, \ldots, 6\}$, and $S = \{\{1,2,3\}, \{3,4\}, \{4,5,6\}\} \subseteq 2^U$, with each $C \in S$ corresponding to one *feature*. Each element $x \in U$ is part of at least one set $C \in S$, thus possessing at least one feature. This example can be encoded as a graph, where the right side "the features" are pre-partitioned into individual cells. By applying the standard partition refinement framework from Section 2.6 with function 2.5, we obtain a partition $\pi \in \Pi(U)$ in which the elements any given cell possess the same features.

Let U be an arbitrary set. Then a covering can be defined as a set $S \subseteq 2^U$, where S^U denotes the power set, such that:

$$U = \bigcup_{C \in S} C$$

The definition is equivalent to a set partitioning without the condition that sets of S must be pairwise disjuct. Each set $C \in S$ corresponds to one *feature*, with the elements $x \in C$ considered to possessing said feature. The goal of the covering version of the slicing strategy is to partition the elements of a given set in such a way, that elements in the same cells all possess the same features. In order to obtain such a partition from a set covering S, we can encode the underlying problem as a graph and use the standard partition refinement framework. Afterwards, the resulting partition can be used to slice the given set as described in Section 5.4.1.

The strategy can be used to e.g. partition constraints according to the types of variables that they contain. It is functionally equivalent to the refinement method "fast" from [9]. The name most likely stems from an algorithm informally described a fast algorithm based on bucket sort to implement such a partition refinement algorithm. Example

5.4.3 Recursive

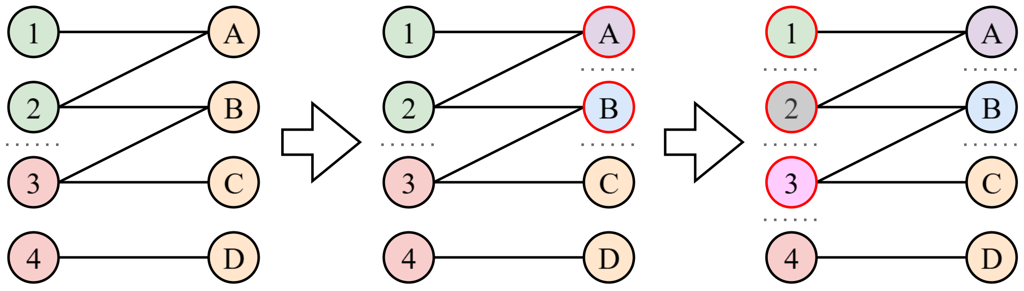


Figure 5.6: An example of a graph for which the partition refinement algorithm takes multiple iterations to yield a stable partition (Function 2.5). Red circles around the vertices highlight changes to the cells in the respective iterations. The algorithm is initialized with $\pi_{init} = \{\{1,2\},\{3,4\},\{A,B,C,D\}\}$. During the first iteration, all vertices on the right side are in the same cell so no changes can happen on the left side. Vertices A, B are not connected to cell $\{3,4\}$. A similar reasoning applies for the second iteration.

The *recursive* strategy is the the only strategy which needs to utilizes the full blown partition refinement framework. The strategy is based on the canonical constraint-variable un-directed graph already shown in Figure 2.2. Each variable and constraint corresponds to exactly one vertex, while an edge between a constraint and variable node exists iff the variable is part of the constraint. More formally, given a set of constraints $C = \{c_1, \dots, c_m\}$ and a set of variables $X = \{x_1, \dots, x_n\}$. Let a_{ij} be the coefficient of variable $x_i \in X$ in constraint $c_j \in C$. Then we can define the graph as follows:

$$G = (V, E) = (X \cup C, \{\{x_i, c_j\} \mid x_i \in X, c_j \in C, a_{ij} \neq 0\})$$

The size of the graph in terms of it's edges and vertices depends entirely to the underlying problem. A partition is obtained by running the algorithm with either Function 2.4 or Function 2.5. Afterwards, the partition is used to slice a given set according to the method described in Section 5.4.1. Note that even thought the generated graph is always bi-partite, it does not necessarily fulfill the conditions mentioned in Section 2.6 for the fast variant of the refinement algorithm, i.e., that all nodes on one of the two sides of the bi-partite graph are all in their own cell. As a consequence, one full execution of the recursive strategy might take orders of magnitude longer than an execution of one of the two slice variants explained previously.

"orders of magnitude" ein wenig übertrieben

5.5 Cutoff Conditions

In order to limit the size of the tree and ensure that we only explore relevant parts of the search space, we propose several conditions the terminate the search early on. The conditions can be divided into two groups:

1. *Local* Conditions, which can be evaluated by only considering information about one *singular* node or its immediate predecessor.
2. *Global* Conditions, whose evaluation requires information about e.g. the precise path to the node, i.e., its position in the tree and therefore information about its ancestors, *or* requires knowledge about other nodes or completely distinct sub-trees.

Note that the following conditions are only *correct* for the strategies presented in Section 5.4, which are all a specialized version of the partition slicing strategy. In the following, let $T = (V, E, U, R, S)$ be a SRT.

5.5.1 Local Conditions

With access only to local information, these conditions are restricted to the information about a single node $v_0 \in V$.

Refined Partition Size

Assuming a proper ground-truth is available or any heuristic information about the potential size $k \in \mathbb{N}$ of the target partition, many found nodes can be excluded from the scoring a-priori. We can terminate the search for any sub-tree rooted at v_0 if $|\text{Cells}_{v_0}| \gg k$. WIP
Here, the motivation is that in any partition π computed in the post-processing in which v_0 is participating, $|\pi| \gg k$ holds.

No Changes

As soon as the algorithm expands a set S with a strategy, we get a partition $\pi \in \Pi(S)$ as a result. If $S \in \pi$, this implies that $\pi = \{S\}$ and the strategy did not refine S any further and we can terminate the search for the current sub-tree.

5.5.2 Global Conditions

As mentioned before, global conditions are more flexible and allow for more involved logic to be executed.

Sub-Tree Duplication

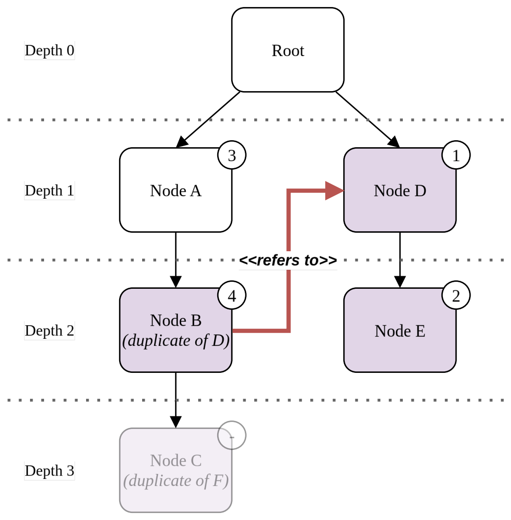


Figure 5.7: Node B is a duplicate of Node E, but the latter was added to the tree before Node B. Thus, as soon as Node B is created this duplication is detected and is never added to the tree. Because this is just an optimization to improve runtime, we have to act *as if* Node B was expanded. Thus, a dummy node is created which *refers* to Node E to keep e.g. the scoring system functional.

If v_0 is expanded by the algorithm, then all cells of the node are expanded in the same way by all strategies that did not run previously. The expansion of a cell by a certain strategy usually results in a new tree node, which is in turn expanded during a future iteration of the algorithm. If a new node that is about to be added already exists somewhere else in the tree, i.e., there is a already an equivalent tree node present, the expansion can stop at that point, since it would lead to the same subtree structure and therefore redundant computation. Here, two tree nodes are considered "equivalent" if they consist of the same cells, i.e., $v_i = v_j \iff \text{Cells}_{v_i} = \text{Cells}_{v_j}$. An example of this condition is illustrated in Figure 5.7.

Most Optimistic Partition Size

Similar to the local condition terminating the search at a node $v \in V$ exceeding a certain number $k \in \mathbb{N}$ of cells, we can terminate the search as it becomes evident that the size of *any* partition containing v and is generated by function 5.5 will exceed k. This *Most optimistic partition size*, i.e., the size of the smallest partition that can be generated using v, can be computed as shown in Function 5.4.

$$\text{MOS}(v) = \begin{cases} 0 & \text{if } v = R \\ |\text{Cells}_{\text{ParentNode}_v}| + \text{MOS}(\text{ParentNode}_v) - 1 & \text{otherwise} \end{cases} \tag{5.4}$$

Because all nodes in the SRT are non-empty, it trivially holds that $\text{MOS}(v) \geq \text{MOS}(\text{ParentNode}_v)$ for all $v \in V$. As soon as $MOS(v) \gg k$, we can terminate the search as all partitions generated using at last one descendant of v will exceed the threshold.

Depth

Under the assumption that most of the "interesting" sets are found early, it could be beneficial to abort the search for a sub-tree as soon as a certain depth has been reached. The depth of a certain node $v \in V$ can be computed using a simple recursion to the root node:

$$\text{Depth}(v) = \begin{cases} 0 & \text{if } v = R \\ 1 + \text{Depth}(\text{ParentNode}_v) & \text{otherwise} \end{cases}$$

This condition should be used with care, as more complex models will most likely require multiple iterations of the algorithm until the SRT has been enriched with enough information to reverse-engineer the original formulation.

Side effects of global conditions

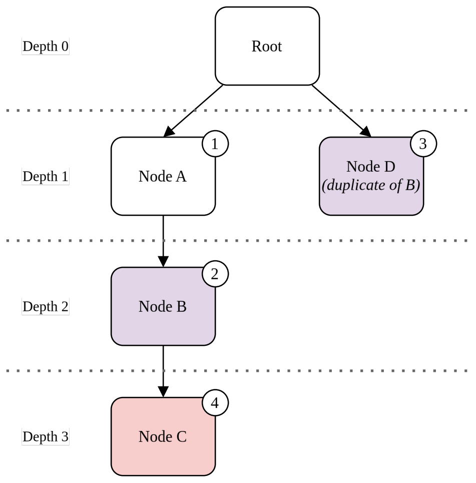


Figure 5.8: a

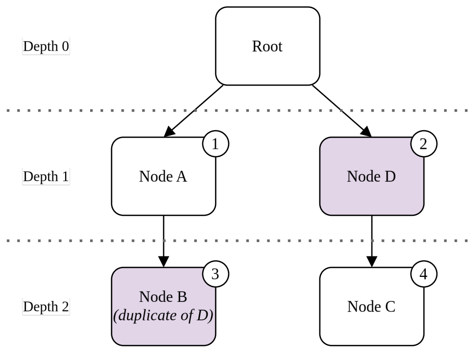


Figure 5.9: a

Figure 5.10: text

Falsches Bild

Conditions such as *Depth* or *Most Optimistic Partition Size* can have unintended side-effects in combination with other global conditions like *Sub-Tree Duplication*. As shown in Figures 5.8 and 5.9 with identical Nodes B and D, the SRT can have a different structure depending on the *order* of node expansion. In the visualizations, nodes were expanded in the order of numbers shown in small circles attached to the nodes. During expansion, the only active cutoff-cconditions were the aforementioned *Sub-Tree Duplication* and *Depth*, which cut off all nodes below or at a depth of 3.

In Figure 5.8, Node B was added to the tree before Node D, thus cutting off the tree after the latter. The sub-tree rooted at Node B was expanded further, until the *Depth*-condition became active. In Figure 5.9, the exact opposite happened. Node D was expanded before Node B, which resulted in the creation of Node C before a sufficient cutoff depth has been reached.

In the case of *Sub-Tree Duplication* and *Depth*, expanding the tree in breath-first order solves the described problem, but side-effects with other conditions are still possible. Thus, depending on the precise set of implemented conditions, one has to evaluate possible side-effects.

5.6 Scoring

Up until this point we have only described how the algorithm refines sets based on different strategies, despite the goal being a list of promising partitions. Before we describe how to select such partitions, we have to define how we actually *get* a list of partitions from the refinement tree.

Let $G = (V, R, E)$ the set refinement tree with vertex set V, designated root node $R \in V$ and edges $E \subseteq V \times V$.
Let $Recombine_k(U_1, U_2, \ldots, U_k) : U_1 \times U_2 \times \ldots \times U_k \to 2^{U_1 \cup U_2 \cup \ldots \cup U_k}, k \in \mathbb{N}$ be a Function defined as follows:

$$Recombine_k(U_1, U_2, \ldots, U_k) = \begin{cases} \varnothing & k = 0 \\ U_1 & k = 1 \\ \{\{u\} \cup r \mid u \in U_1, r \in Recombine_{k-1}(U_2, U_3, \ldots, U_k)\} & else \end{cases}$$

More informally, $Reombine_k$ takes a total of k arbitrary sets $U_1, \ldots, U_k$ and computes a set containing all possible k-tuples $(u_1, u_2, \ldots, u_k)$ with u_i restricted to elements from U_i. Thus, the set of *all* partitions constructible from a sub-tree rooted at vertex $v \in V$ can be computed using Function 5.5.

$$Partitions(v) = \begin{cases} Sets(v) & \text{if } v \text{ is leaf node} \\ \left\{Recombine_{|Sets(v)|}\right\} & else \end{cases} \tag{5.5}$$

It is clear that setting $v = R$ gives the set of all possible partitions.

WIP

Add simple proof

One can deduce from the definition of Function 5.5 that the number of partitions depends on the depth and maximum out-degree of any given node of the tree. Therefore, it will be of vital interest to only consider "interesting" sets. Ideas and possible implementations are discusses in Chapter 6.

Example

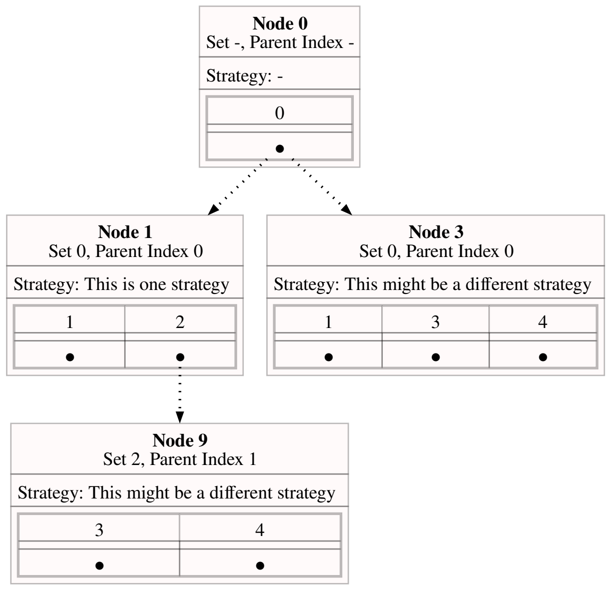


Figure 5.11: A sample SRT with four nodes. Node 1 has one descendant, so the recursive function will generate multiple possible partitions because of that.

In order to illustrate the generation of partition from a given SRT, we use the tree shown in Figure 5.11 as an example. In the following, we abbreviate a node "Node i" with N_i. Applying the definition of Function 5.5 to the SRT, we have start at the root node with identifier 0.

$$Partitions(N_0) = Recombine_1$$

Because the root node only consists of one cell, we just take the union of $Partitions(N_1)$ WIP and $Partitions(N_3)$. For the latter, we can compute the result easily, because N_3 is a leaf node. Thus, the set of all possible ways of generating partitions for cell 0 in this sub-tree consists of the single partition $\{\{1,3,4\}\}$ For N_1, we have to execute the recursion twice:

1. The first possibility to partition cell 0 is $\{1,2\}$.
2. In addition, we have another option to partition set 2: $\{3,4\}$, which is possible because of Node 9.

Thus, we can partition cell 0 in this sub-tree in two ways: $\{\{1,2\},\{1,3,4\}\}$. The union of the left and right sub-tree results in three possible ways to partition cell 0, finishing the partition generation phase.

5.6.1 Constraint Names

One important piece of information that usually reflects the modelers *intent* when it comes to the semantic difference are the names of the constraints or variables, in the following abbreviated with just “names”. As discusses in Section 4.2.1, names usually consist of multiple parts, with one part containing all the information related to *semantics*. Extracting this part of the name can be done heuristically using Algorithm 10. The heuristic consist of four major steps:

1. Remove *modifies* which are usually enclosed in either brackets or some other special character combination which consists of an *opening* and *closing* character.
2. “Normalize” the string by replacing any non-alpha-numeric separators such as spaces, tabs, . . . with a single character c such as “_”.
3. Split the string according to c and try to detect the part with the relevant semantics.
4. *(Optional)* Remove any left-over non-alpha characters such as numbers.

Step 4 is explicitly marked as optional, because sometimes numbers are integral part of the name, sometimes they are not. Both of these cases can actually be observed in the same model [1]. For this model in particular, the constraints “cut1” to “cut4” are of interest. The constraints with the prefix “cut1” and “cut2” are structurally identical, i.e, the they all share the same number of non-zeros and the same type of variables, while “cut3” and “cut4” do not. Here, “cut1” and “cut2” would have to belong to the same group, while “cut3” and “cut4” belong in a group their own. With the heuristic as is, this case is not detected with or without step 4 because additional information about the actual structure of constraint is required.

Note that the Algorithm makes some implicit key assumptions, such as:

- That the name actually carriers any semantically relevant information at all.
- The names adhere to a “common” format, i.e., the modifier comes *after* the semantics part. Here, constraint names such as “capacity_bin_k”, “out_flow_k” qualify, but the same names in reverse do not.

[1] StrIPlib UUID: d48c9568-e0ea-4c77-9a3b-b7f4dc530d5f

These assumptions proved reasonable for almost all models observed in practice which had proper names available. Still, there cases where this approach fails .

Give MI-PLIB examples

Algorithm 10

Input: Name of a constraint

Output: Relevant semantics of the constraint name if the name adheres to well known naming scheme

```
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. ], }, ), . . .
        $name_{new} \leftarrow name_{old}$ without characters in range $[start, end]$
    until $name_{new} = name_{old}$
end function
```

WIP

As soon as the semantic part of all names is extracted, constraints and variables can be grouped based on this information. Possible post-processing steps might include an application of the mentioned Levenshtein-Distance from Chapter 4. Even thought the Algorithm has quadratic time complexity and is therefore unlikely to be used with the full set of original constraint or variable names, a reasonable assumption is that

$$\{\, \textsc{ExtractSemanticPart}(Name(c)) \mid c \in Cons \,\} \ll \{\, Name(c) \mid c \in Cons \,\}$$

Analogously, the assumption is made for variable names as well. Thus, an Algorithm for the Levenshtein-Distance, even under worst-case considerations, might be applicable and thus resulting in a better partitioning.

5.6.2 Ground Truth based

As discusses in the previous Sections, the algorithm starts with a set in which all constraints or variables are in the same cell. In order to *guide* the search towards a plausible semantic partitioning, the existence of a ground-truth partition which approximates the desired partition reasonably well can be used to terminate the search or select promising partitions after the search-space has been explored. Terminating the search can, in practice, be very useful, because preventing the algorithm from expanding a node not only prevents unnecessary work being done for this particular node, but also prevents the generation of the entire sub-tree below; reducing the number of potential candidate partitions and overall runtime and space requirements of the algorithm.

Potential sources for a suitable ground-truth partitions include

- Constraint names using the heuristic from Section 5.6.1.
- Variable information, i.e., given a ground-truth of semantic groupings of variables, one could obtain a corresponding ground-truth partition for constraints by assigning two constraints to different groups iff they contain different kinds of variables according to the ground-truth.

Without the availability of a reasonably ground-truth, it is currently unclear how to know when to terminate the search and more importantly, what structured the desired target partitions should have. Here, the term "structure" refers to all relevant properties of the target partition, including but not limited to, the number and size of the cells.

Assuming a reasonable ground-truth has been obtained, a partition-comparing score such as the Rand-Index already discussed in Section 2.8 can be used a number of promising candidates. Furthermore, the refinement can be terminates for sets which are already *homogeneous* according to the ground-truth partition. The idea being is that by refining the set further we do not gain any new information.

completene
missing

5.6.3 Connected Components Score

The ground-truth based scoring from Section 5.6.2 scores candidates based on a partition π, which is assumed to resemble a semantic grouping reasonable close to the real partitioning in order to "guide" the search and find promising candidates. In case this assumption is not true, it can be expected that the selected partitions provide no useful information about the model.

In the following, let $A \in \mathbb{Q}^{m \times n}$ be a constraint matrix, $G = (V, E)$ with $V = \{1, 2, \dots, m\}$ and $E = \{\{u, v\} \in V \times V \mid \exists i : A_{ui} \neq 0 \wedge A_{vi} \neq 0\}$ a graph with the constraints as its vertices. There exists an edge between two constraints iff they share at least one variable with non-zero coefficient. Furthermore, let $cc(v) : V \to 2^V$ be a function mapping every vertex to its connected component in G. It is a well known result from graph-theory that every graph can be decomposed into its connected components, so the function is unique. cite

In order to get to a score based on connected components, we define the relation $\sim_\pi = \{(B_i, B_j) \in \pi \times \pi \mid (cc(B_i) \mid cc(B_j)) \vee (cc(B_j) \mid cc(B_i))\}$, where $a \mid b$ with $a, b \in \mathbb{N}$ denotes the standard divisibility relation defined on natural numbers. Given a partition $\pi = \{B_1, B_2, \dots, B_k\}$, we can compute a "connected components score" as follows:

$$\text{connectedScore}(\pi) = |\pi / \sim_\pi|$$

The score corresponds to the amount of connected components of "different" sizes. Here, two sizes $a, b \in \mathbb{N}$ are considered different, if a is not divisible by b *and* vice-versa. motivating example consequen

6 Implementation

In the context of this work, the algorithm including all described classifiers, scores and strategies from Chapter 5 were implemented and integrated into GCG, with the only exception being the unordered variant of the *voting classifier* [1]. Building upon the algorithmic descriptions of Chapter 5, we want to discuss how such an algorithm can be efficiently implemented and practice and how the component is integrated into GCG as a detector. In the actual code, all shown algorithms and data structures were implemented and tested in `C++`. The Chapter is divided into multiple sections, each describing a different aspect of the implementation:

1. Section 6.1 contains a high-level overview about the architecture of the detector and its relation with other components in GCG.
2. In Sections 6.1 - 6.3, we will give a brief overview about a few custom data structures required to implement the refinement more efficiently and how the tree is represented in memory.
3. Sections 6.4 and 6.5 conclude this Chapter with an extension to the algorithm to improve the overall runtime on modern hardware.

We only discuss the most noteworthy aspects of the implementation, thus excluding the following components:

- All classifiers as described in Section 5.3.
- "Glue code" putting together all individual pieces of the implementation which are discussed in the following sections, i.e., the actual code and logic that combines the parts to one coherent algorithm.
- Concrete pieces of `C++`, as we do not require the reader to know this particular programming language on a sufficient level.

[1] In practice, almost all tested models obeyed the precondition of coherent constraint blocks

6.1 Architecture

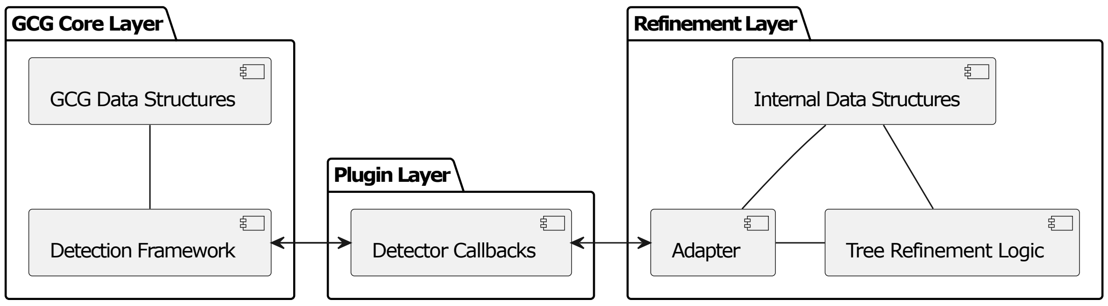


Figure 6.1: text

As mentioned in the introduction to Chapter 5, the approach is implemented as a *Detector* in GCG, contrary to the fact that the algorithm outputs a one or multiple partitions of constraints or variables. Generating such partitions is more aligned with the concept of a *classifier* from Section 4.2, but this comes with an important drawback. Because we want to incorporate information about variables for partitioning constraints and vice-versa for constraints, we have to delay the execution of the algorithm to a point in time *after* the classification has finished, i.e., all relevant data is available. Circumventing this problem by potentially implementing the algorithm as a classifier and assigning the lowest priority possible to it is not an option as well, because this would introduce additional maintenance overhead in case changes to the overall classification/detection framework are made. An implementation as a detector *ensures* that all classification step are done beforehand.

An overview about the relationship between the GCG and the detector is shown in Figure 6.1. When implementing an detector, GCG provided a set of callbacks that have to implemented such as

- Set-up/Tear-down, i.e., for allocation and deallocation of data-structures
- A handler for propagation, which takes a partial decomposition and assigns all or a subset of the remaining open constraints to either a block or the master. This concept was already shown in Figure 4.3.

The callbacks take GCG-internal data structures as input and must provide the result as such. In order to ensure better maintainability, the logic realizing the tree refinement should be mostly *independent* of the concrete framework it is being used in. Furthermore, relying on custom data structures increases control about runtime and space considerations. This decoupling is being realized by an *Adapter*, as shown in Figure 6.1, which translates between the two "worlds" of data-structures and ensures compatibility.

Wording

6.2 Data Structures

The implementation of the central tree refinement algorithm uses the following core components:

- *Strategies*: As explained in Section 5.4, strategies are responsible for refining cells. Each strategy is implemented as a separate data-structure which share a common interface. Each strategy is uniquely identified by its name and splitter-function $f_{strat} : U \to \Pi(U)$.
- *Rules*: A rule is responsible for deciding whether a certain strategy *is allowed* to expand a SRT node. Rules *can* be used to implement local cutoff-conditions such as *Depth* or limit the set of applicable strategies dynamically. Conceptually, they can be seen as a predicate taking a node and strategy as input. In contrast to strategies and scores, rules are purely an implementation detail and are not required for a functioning algorithm, as cutoff-conditions might be implemented in a less dynamic way.
- *Scores*: As the name implies, scores are responsible for assigning each found candidate partition a numeric value. In addition to that, each scores defines a total order on the numeric values, i.e., for some scores higher values correspond to "better" partitions, while for others it might be the opposite. Similar to strategies, a score is uniquely defined by its name. Each score provides a function $g : \pi \mapsto \mathbb{R}$, mapping a partition to a number, and an order on these values.
- *Auxiliary data structures*, such as a thread-safe queue, an efficient hash-table for representing partitions in-memory and the representation of a single tree node of an SRT.

Representing a Tree Node

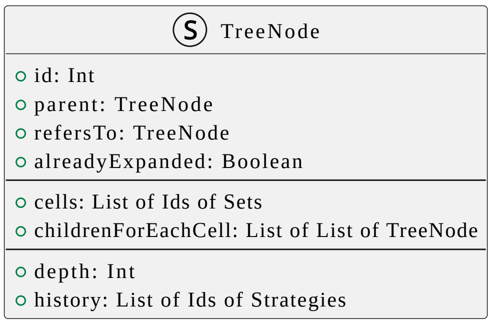

S TreeNode
id: Int
parent: TreeNode
refersTo: TreeNode
alreadyExpanded: Boolean
cells: List of Ids of Sets
childrenForEachCell: List of List of TreeNode
depth: Int
history: List of Ids of Strategies

Figure 6.2: a

One of the *central* data structure is the SRT, for which a shortened version is shown in Figure 6.2. To simplify data access to all information about a singular node, we store all data in one centralized object. This allows other parts of the algorithm to access all necessary data related to a node at once, which consists of the following properties:

- Basic information identifying a node, such as its *id*.
- The *parent* node, including a copy of the *history* of all strategies that were applied before.
- The *cells* the node consist of including all *children* that were created after expansion of each cell.
- Information about the nodes *depth* in the tree and a pointer to a different tree node to support certain cutoff-conditions such as *Subtree Duplication* more easily (see Figure 5.7).

The tree can be accessed through a reference to the root node R of the SRT. Properties like *cells* and their efficient storage will be of vital interested in Section 6.3.

6.3 Duplication Prevention

In order to keep the memory consumption of the SRT, which represents the explored search space, as low as possible within practical bounds, we propose two ways of to achieve this goal:

1. A simple approach to storing the actual partitions and its cells in memory by leveraging knowledge about previously generated cells.
2. We prevent the generation of *identical sub-trees* by the algorithm to explore the same search space without additional computational effort.

We propose two data structure for storing and indexing of individual cells and tree nodes as shown in Figure 6.3. Both data structures require the following basic operations:

- Basic operations including adding, deleting and containment checks via. `add(·)`, `remove(·)` and `contains(·)` respectively.
- An operation to get, based on some cell or tree node object, the exact duplicate stored inside the data structure. As shown in Figure 6.3, both data structures realize this via. `getRepresentative(·)`
- A function `addIfMissing(·)` adding the tree node if it doesn't exist already. These two actions are performed *atomically*, i.e., to an outside observer the method behaves as if it executes both operations at once.

While precise implementation of the hash table is not of interest here, we assume that processing common queries should be done reasonably efficient.

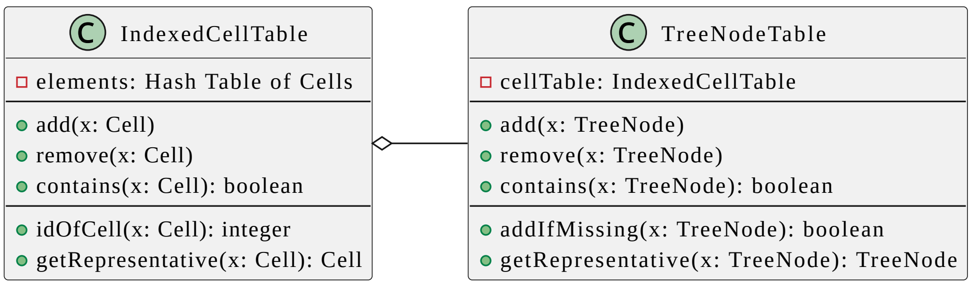

IndexedCellTable
elements: Hash Table of Cells
add(x: Cell)
remove(x: Cell)
contains(x: Cell): boolean
idOfCell(x: Cell): integer
getRepresentative(x: Cell): Cell
TreeNodeTable
cellTable: IndexedCellTable
add(x: TreeNode)
remove(x: TreeNode)
contains(x: TreeNode): boolean
addIfMissing(x: TreeNode): boolean
getRepresentative(x: TreeNode): TreeNode

Figure 6.3: text

In the following, we will propose two ways of implementing appropriate hashing functions to realize such data structures. With these, we can reduce memory consumption *and* implement the sub-tree duplication mechanism mentioned in Section 5.5 efficiently.

Hashing for single cells

In order to keep the *size* of the SRT in terms of its actual footprint in RAM as small as possible, we propose a simple solution based on hashing. As soon as a strategy refines a set S, we get a partition $\pi = \{A_1, A_2, \dots, A_k\} \in \Pi(S)$. The cells of all found partitions including $A_1, A_2, \dots, A_k$ are stored in a central data structure and assigned a unique index each. Cells that are already stored in the data structure are *not* added again to reduce memory consumption. The duplication check for a cell $A = \{o_1, o_2, \dots, o_n\}$ is done by computing $x = \textsc{HashList}(A)$ and probing the data structure for x. If no match was found, we add the set to the data structure. If a match was found, we abort.

Hashing for SRT Nodes

Based on $\textsc{HashList}$, we can define a hash function for nodes of a given SRT $T = (V, E, U, R, S)$ in a similar manner as for individual cells:

$$\mathrm{HashTreeNode}(v) = \textsc{HashList}(\textsc{Sort}(\{\mathrm{CellId}(C) \mid \forall C \in \mathrm{Cells}_v\}))$$

The function $\textsc{CellId}$ probes the data structure mentioned in the previous Section for the unique id of the cell. This way, two nodes with identical cells are also assigned the same sequence of ids. Note that the extracted node-ids are sorted before the hashing, because the output of $\textsc{HashList}$ is dependent on the *order* of the elements in the list. It can be expected that any given node only consists of a small number of cells and function $\textsc{HashList}$ can be implemented very efficiently, $\textsc{HashTreeNode}$ can be as well. Thus, by keeping a table mapping hash values to its associated nodes in memory, we can check for duplicates without a linear search through V. In case a duplication of tree nodes is detected, we have to test for equality of the two nodes to account for hash collisions. Here, testing for equality of two nodes $v_1, v_2 \in V$ can be done by evaluating $\mathrm{Cells}_{v_1} = \mathrm{Cells}_{v_2}$ for this single pair.

6.4 Concurrency

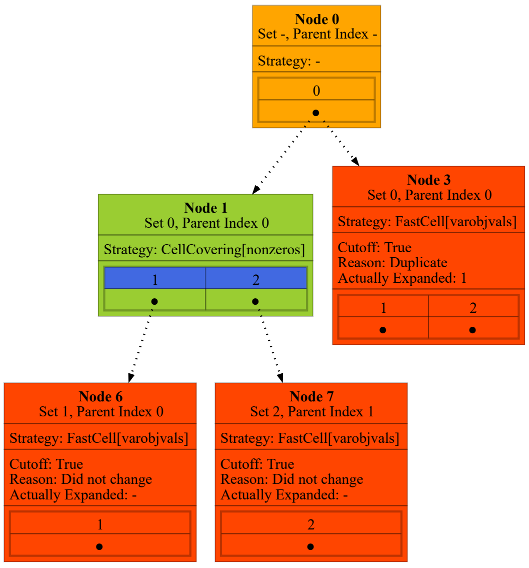


Figure 6.4: text

In addition to the optimizations via. hashing and conditional termination of the search mentioned in Sections 5.5 and 6.3, which primarily focused on reducing memory requirements, we now propose a method to reduce the actual runtime. Based on the description of the overall algorithm from Chapter 5, we recall two important properties:

1. When no global cutoff conditions are being used, a node of the SRT is expanded is always expanded in the same way regardless of its position in the tree.
2. The set Cells$_v$ for all nodes only depends on the parent cell.

Wording

Therefore, we can parallelized the algorithm rather easily by using a centralized queue which is explained in more detail in Section 6.5. In general, a queue is a data-structure providing access to a set of mutable elements in first-in first-out order. For the purposes of this Section, we require the existence of three methods for accessing the elements:

1. `push(·)`, adding an element to the end of the queue.
2. `pop(·)`, returning the last element of the queue and removing it. If the queue is empty, this method should block execution of the calling thread until an element was added via. `push(·)`.

3. `size()`, returning the amount elements still remaining in the queue.

The queue $Q = \{q_1, q_2, \dots, q_n\}, q_i \in V$ for a SRT $T = (V, E, U, R, S)$ is initialized with R. Afterwards, a fixed amount of *workers* is started that will perform the following steps until an *termination signal* is received or no further progress can be made. A high-level overview of the logic executed by a single worker is shown in Algorithm 11. An exact implementation for the termination signal is omitted, but a proposal can be found in Section 6.5. Because of the multi-threaded nature of this approach, the queue and all auxiliary data-structures used *must* provide appropriate synchronization to prevent race-conditions.

Algorithm 11 Algo

Input: An instance *treeNodeTable* of the Tree Node Table from Figure 6.3, a *queue* shared among all workers, a list of *strategies*.
Output: None

```
function WORKER(treeNodeTable, queue, strategies)
    while true do
        if termination signal received then
            return                                         ▷ Shutdown worker
        end if
        $q \leftarrow$ POP($queue$)
        for $cell \leftarrow \text{Cells}_q$ do
            for $f_{\text{strat}} \in strategies$ do
                if the pair $(q, strat)$ is forbidden by some rule then
                    continue
                end if
                $refined \leftarrow f_{\text{strat}}(cell)$
                if $refined$ violates a cutoff-condition then
                    continue
                end if
                $treeNode \leftarrow$ create new tree node for $refined$
                $added \leftarrow$ ADDIFMISSING(treeNodeTable, treeNode)
                if $added$ is true then             ▷ Only add to queue if no duplicate
                    PUSH(queue, treeNode)
                end if
            end for
        end for
    end while
end function
```

6.5 Queue

In order to design a queue with the requirements mentioned in Section 6.4 that support access to its data from multiple threads, we use a “standard” queue and restrict access to it via. two synchronization primitives which are explained in more detail in Section 2.5:

1. A *mutex*, which is used to protect the queue itself from simultaneous access from different threads.
2. A so called “condition variable”, which allows threads to suspend and resume execution until some predicate or *condition* on shared data is satisfied. Here, the shared data in question is the queue itself. The predicate evaluates to true iff the queue is non-empty, i.e., a calling thread is blocked until there is at least one element in the queue.

For our proposed implementation, we will use two condition variables with two different predicates:

1. A variable `queueNonEmptyCond`, which blocks workers until the queue is non-empty.
2. The variable `workersWaitingCondVar`, which blocks workers until the queue *is* empty and all workers are being blocked. This variable is being to wait until the correct moment to terminate the search as explained below.

Algorithm 11 requires an addition mechanism which we integrate into the queue data structure itself: a way to send all workers a termination signal. We will only use the signal to notify all threads to shut down until all possible nodes have been expanded, that is, until all k workers have called Pop and were suspended because the queue is empty. If this condition is met, we add k instances of a special element Ω to the queue, unblocking the workers. As soon as a worker pops an element from which corresponds to Ω, it terminates. These special elements are often referred to as “poison pills” and are used to communicate with workers without modifications to the interface of the data structure. cite

Algorithm 12 text

```
function PUSH(queue, element)
    LOCK(mutex)                                          ▷ Protect queue
    PUSH(queue, element)
    UNLOCK(mutex)
    NOTIFY(queueNonEmptyCond)    ▷ Notify other threads that queue size changed
end function
```

Algorithm 13 text

```
function POP(queue)
    LOCK(mutex)

    $numWaiting \leftarrow numWaiting + 1$
    NOTIFY(workersWaitingCondVar)
    WAITFOR(queueNonEmptyCond, mutex)
    $numWaiting \leftarrow numWaiting - 1$

    $element \leftarrow$ POP($queue$)
    UNLOCK(mutex)
    return $element$
end function
```

Algorithm 14 text

```
function INJECTPOISON(queue, numWorkers)
    LOCK(mutex)
    WAITFOR(workersWaitingCondVar, mutex)
    repeat
        PUSH(queue, $\Omega$)
    until $numWorkers$ element have been added
    UNLOCK(mutex)
end function
```

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

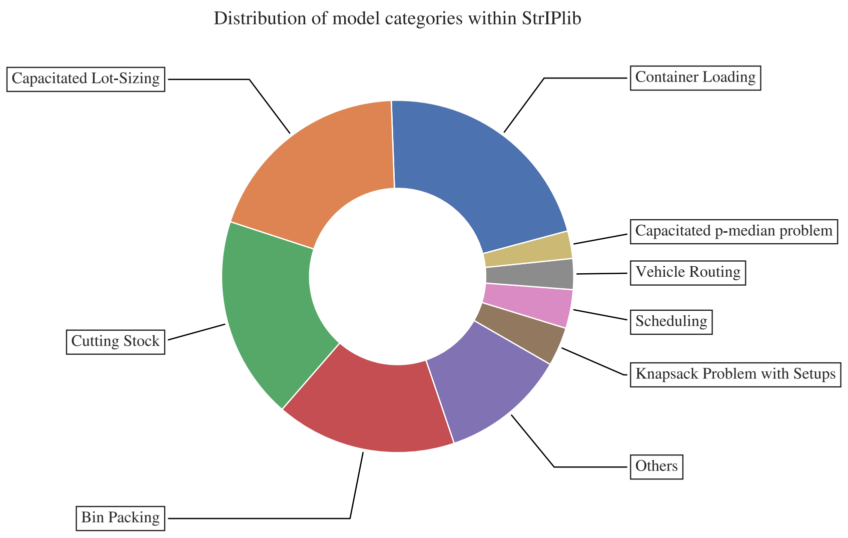


Figure 7.1: The distribution of different categories of model within strIPlib. Most problems are part of "common" categories like Bin-Packing, Scheduling and Routing. The category "Others" includes e.g. Fantasy Football, Train Scheduling and different types for which only a small number of model files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

Literaturverzeichnis

[1] *Branch & Price.*

[2] Christopher Wellons. *Hash Function Prospector.* 09/26/2025. URL: https://github.com/skeeto/hash-prospector.

[3] Upadhyay, D. et al. “Investigating the Avalanche Effect of Various Cryptographically Secure Hash Functions and Hash-Based Applications”. In: *IEEE access : practical innovations, open solutions* 10 (2022), pp. 112472–112486.

[4] Boost. *Boost C++ Libraries.* Boost. URL: http://www.boost.org/.

[5] “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).

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

[7] 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).

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

[9] 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).

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

[11] 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.

[12] 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).

[13] “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).

[14] 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.

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

[16] 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).

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

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

[19] 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.