Improving Scheduling by Learning

Andreas Schutt

Submitted in total fulfilment of the requirements

of the degree of Doctor of Philosophy

June 2011

Department of Computer Science and Software Engineering
The University of Melbourne

Produced on archival quality paper

To my parents

Abstract

Scheduling problems appear in many industrial problems with different facets and requirements of a solution. A solution is a schedule of a set of activities subject to constraints such as precedence relations and resource restrictions on the maximum number of concurrent activities. This dissertation presents new deductive techniques for precedence and cumulative resource constraints in constraint programming to improve solution processes of combinatorial scheduling problems, in particular, and combinatorial problems, in general.

Due to their intractability, many schedulers either solve a simplified problem or are tailored to a specific problem, but are inflexible with respect to changes in the problem statement. Constraint Programming (CP) offers a powerful framework for combinatorial problems which also tackles the demand of flexibility of changes in the problem statement due to a strict separation of problem modelling, search algorithms, and high-specialised deduction techniques. Moreover, recent advanced CP solvers such as lazy clause generation (LCG) additionally include sophisticated conflict learning technologies. Their efficiency depends, amongst other things, on reusable explanations formulated by deductive algorithms.

Unit two variable per inequality (UTVPI) constraints are one of the largest integer constraint class that is solvable in polynomial time. These constraints can model precedence relations between activities. A new theoretical result about reasoning of UTVPI systems is shown, based on that, new incremental deductive algorithms are created for manipulating UTVPI constraints including satisfiability, implication, and explanation. These algorithms are asymptotically faster on sparse UTVPI systems than current algorithms.

Cumulative constraints enforce resource restrictions in scheduling problems. We show how, by adding explanation to the cumulative constraint in an LCG solver, we can solve resource-constrained project scheduling problems far faster than other comparable methods. Furthermore, a complete LCG-based approach including cumulative constraints is developed for an industrial carpet cutting problem. This approach outperforms the current incomplete method on all industrial instances provided.

Declaration

This is to certify that

(i) the thesis comprises only my original work,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of table, maps, bibliographies, appendices, and footnotes.

Signature_____

Date_____

Preface

The research herein resulted from numerous discussions with my supervisor, Prof. Peter J. Stuckey, and his replacement, Prof. Mark G. Wallace, during his sabbatical. Chapters 4 and 5 present work that are based on a lazy clause generation solver which was implemented by Dr Thibaut Feydy. He helped me to fix bugs that I discovered in this solver. Dr Andrew R. Verden collaborated on the industrial carpet cutting problem (Chap. 6) by providing the industrial instances and giving insights into the current solution method.

The main part of this dissertation has already been published or is under review in the following papers.

- Incremental Satisfiability and Implication for UTVPI Constraints. *INFORMS Journal on Computing* **22**(4) 514–527, 2010.
  - Joint work with Peter J. Stuckey.
  - Part of Chap. 3.
- Why cumulative decomposition is not as bad as it sounds. Ian Gent, ed., *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, *LNCS*, vol. 5732, 2009. Springer-Verlag, 746–761.
  - Joint work with Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace.
  - Part of Chap. 4 and 5.
- Explaining the cumulative propagator. *Constraints* **16**(3) 250–282, 2011.
  - Joint work with Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace.
  - Part of Chap. 4 and 5.
- Solving RCPSP/max by Lazy Clause Generation. Submitted to *Journal of Scheduling*. Earlier version available at `http://arxiv.org/abs/1009.0347`.
  - Joint work with Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace.
  - Part of Chap. 5.

- Optimal Carpet Cutting. Jimmy Lee, ed., *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, *LNCS*, vol. 6876, 2011. Springer-Verlag, 69–84.
  - Joint work with Peter J. Stuckey and Andrew R. Verden.
  - Part of Chap. 6.

Acknowledgement

First and foremost, I would like to thank my supervisor Prof. Peter J. Stuckey. Without his excellent support, patience, and guidance none of this thesis would have been possible. To him I owe most of my understanding of Boolean satisfiability and satisfiability modulo theories solving. Thanks are also due to my advisory committee, Prof. Mark G. Wallace and Dr. Zoltan Somogyi, for their availability and comments, especially Mark who supervised me during the Peter's sabbatical.

This dissertation has been funded by The University of Melbourne and National ICT Australia, Victoria, as a part of an enhanced PhD program. In addition, I am grateful to National ICT Australia for giving me the opportunity to attend conferences and visit a leading laboratory in my field at L'École des Mines de Nantes, France.

I am indebted to IF Computer GmbH and Dr Andrew R. Verden for providing me not only industrial data for their carpet cutting problem, but also sharing their knowledge and results with me.

I am very thankful to all members, past and current, of the Victorian G12 team in National ICT Australia in which I worked: Peter, Mark, Zoltan, Prof. Kim Marriott, A/Prof. Maria Garcia de la Banda, Dr Sebastian Brand, Mark Brown, Dr Leslie De Koninck, Dr Thibaut Feydy, Julien S. Fischer, Dr Alexander D. Stivala, Dr Ralph Becket, Paul Bone, Dr Gregory J. Duck, Dr Nicholas Nethercote, Dr Jakob Puchinger, and Dr Horst Samulowitz. It has been a wonderful and dynamic group to work with. I am glad for all their help and the discussions, thesis related or not, during my doctoral studies, especially Julien, Leslie, Sebastian, and Thibaut. Furthermore, I am greatly indebted to Alexander and Sebastian who were very enthusiastic in the proof-reading of this thesis; without them the thesis would look worse.

I was introduced to the worlds of Constraint Programming and research when I was working for the group PlanT at Fraunhofer FIRST, Berlin, Germany. This was a great time and experience for me which influenced my decision to continue research in this area. I am greatly indebted to Dr Armin Wolf who was my Master thesis supervisor and introduced me the cumulative resource constraint. A special thanks are also due to Prof. Ulrich Geske, Dr Hans-Joachim Goltz, and Dr Dirk Matzke as well as the other members of the former group.

Without my friends and colleagues from the Computer Science and Software Engineering Department the PhD life would not be so funny and happy; you were the spice in my soup. I am especially grateful to Dr Jubaer Arif, Raj K. Gaire, Tanzima Hashem, Dr Marco A. S. Netto, Dr Hua Jie "Jason" Lee, Dr Zi Zhao "Robert" Lieu, Adel Nadjaran-Toosi, Dr Lei Ni, Dr Suraj Pandey, Dr Mukaddim Pathan, Dr Ziyuan Wang, Archana Sathivelu, Dr Christian Vecchiola, Dr Srikumar Venugopal, William Voorsluys, Guey Yong "Camella" Chee, Châu Trần Tú Anh, and Vũ Thị Ánh Tuyết also for their help and advice in all matters of life.

I wish to give whole-hearted thanks to my parents and brothers for their amazing support and dedication over all stages of my life. In particular, I am indebted forever to my parents for their fight for my timely education. Without their fight and belief in me I may not have been able to attend any university. *Vielen Dank für Eure selbstlose Unterstützung. Ihr seid großartige Eltern und Brüder.*

I thank the examiners for their work and comments which have helped to improve the content of the thesis.

Finally, I would like to thank my friends in Germany, in particular Nicole, René, and Sven, and my fiancée Tiên for their strong support, especially Tiên for her patience and understanding during these years. No-one is happier to see this dissertation completed than she is.

Andreas Schutt
Melbourne, Australia
June 2011.

Contents

List of Figures

List of Tables

1 Introduction

Scheduling is a broad area with many facets and different requirements on a solution, e.g. robustness, optimality, or fast-to-generate. This dissertation focuses on the optimal solution generation of combinatorial scheduling problems involving scarce cumulative resources. The techniques developed herein are tied to particular constraints that occur in such scheduling problems. Hence, these techniques are only one part of the generation process. Moreover, their usage is not restricted to scheduling problems. They can be applied to any combinatorial problem that involves these constraints, *i.e.*, the techniques are versatile. For instance, we apply one technique also to an industrial placement problem.

Due to the intractability of these combinatorial problems, no complete method—methods that find and prove optimal solutions—exists that can solve any problem in polynomial time, unless $P = NP$. But this does not mean that all problems are not solvable in polynomial time or, more importantly, cannot be solved in a reasonable amount of time. Research focusses on improving solution methods, so that more and larger problems can be solved in a timely manner.

Although complete methods might not be applicable to solve large-scale problems, they can be combined with incomplete methods in order to optimally solve smaller subproblems. Moreover, they can be truncated in time, *i.e.*, aborted after a certain amount of runtime has elapsed, or used for the evaluation of incomplete methods. More importantly, only complete methods can prove the infeasibility of a problem.

1.1 Current Approaches

Complete and incomplete methods for combinatorial scheduling problems have mostly been proposed from Artificial Intelligence, Constraint Programming (Cp), and

Operations Research communities.

Incomplete methods are mainly meta-heuristics, e.g. ant colony optimisation, evolutionary algorithms, simulated annealing. Depending on the meta-heuristic, they start with one solution or more. These solutions are improved in an iterative process that moves from the current solution to another solution in the neighbourhood. Consequently, the meta-heuristics can converge to a local optimum. Thus, they cannot be guaranteed to find an optimal or a close-to-optimal solution.

The efficiency of these methods depends on the encoding of solutions and a fine-tuning of the parameters of the meta-heuristics. Typically, the encoding exploits problem-specific characteristics and the fine-tuning is made on sample instances of the problem. Therefore, they may be invalid if the problem changes, *i.e.*, if constraints are added or removed from the problem.

Overall, these methods are fast algorithms to obtain good solutions, but they are inflexible for changes in the problem.

In contrast to the incomplete methods, complete methods exhaustively explore the search space. Thus, they can prove the infeasibility of a problem and find the optimal solution. But they may not solve the problem in a timely manner, because of the intractability of the problem.

The search is based on a search algorithm which stepwise builds a solution. Usually, scheduling solutions enhance the search algorithm with dominance rules in order to drastically reduce the search space. Dominance rules define a partial order on the set of solutions. Instead of exploring all solutions the search algorithm only explores the dominant solutions. These rules are normally deduced from the problem considered and tightly connected to the search algorithm. Hence, these rules may not be applicable if the search algorithm changes or invalid if the problem changes.

Generic methods are flexible to changes in the problem statement, but they are often viewed not to be competitive methods with tailor-made methods. One reason is that they do not deploy learning methods or efficient learning methods. The techniques developed herein are embedded in solvers that use efficient learning technologies and follow the CP paradigm of a strict separation of problem modelling, search algorithms, and highly-specialised deductive techniques which prune parts without any solution from the search space.

This strict separation results not only in a flexible framework in which an alternation of the problem statement can be easily accommodated, but also in efficient optimal solving of a large variety of problems. Furthermore, the separation allows the application of the developed techniques to any problem that involves the corresponding constraints.

1.2 Constraint-based Solving with Learning

A constraint-based solving method assumes a problem consisting of variables, their respective domains, *i.e.*, values they can take, and constraints over variables. The goal is to determine if the problem is solvable, or to find an optimal solution if an objective function is given. A solution is an assignment of values from the domains to the respective variables, so that the assignment meets all constraints.

The solution process can be seen as a traversal of a search tree (decision tree). A basic search algorithm determines how to traverse the tree from the root node to the leaves. In each node, the search algorithm branches over the problem, *i.e.*, splits the problem into several subproblems, and decides which branch (subproblem) to explore next. If the search reaches a leaf then it either found a solution of the original problem or detected a conflict, *i.e.*, it proved the infeasibility of the subproblem. In the latter, the search backtracks to the previous node and explores another unexplored branch from the node. The search continues until a solution is found or the infeasibility of the original problem is proven by fully exploring the search tree.

This basic search scheme is enriched with constraint propagation and conflict learning for some advanced constraint-based solvers.

Constraint propagation is a repeated execution of deduction techniques (called constraint propagators) for each constraint, which remove inconsistent values from the domains of variables, until no further deduction can be made. These constraint propagators are highly-specialised deduction techniques which are independently run without knowledge of other constraint propagators. These domain reductions can lead to pruning of some inconsistent subtrees, *i.e.*, trees containing no solutions, from the tree rooted at the current node. Thus, the search does not have to explore those subtrees. The constraint propagation is performed in each node before the search branches.

Conflict learning is another technique to cut inconsistent parts of the search tree, but it is more complex than constraint propagation. If the search hits a conflict node then conflict learning is initiated. It deduces a new constraint (called a nogood) taking account of previous search decisions and/or domain reductions performed by constraint propagators. Then this nogood is temporarily or permanently added to the set of constraints of the original problem. Hence, it is propagated during constraint propagation in the remaining search.

If conflict learning considers domain reductions then each constraint propagator is required to explain its propagation. But it should be explained in an efficient way, and in the strongest possible way, otherwise the inferred nogood is weak, *i.e.*, its

propagation only causes limited reduction of the remaining search.

In addition, advanced solvers collect information during conflict learning in order to drive a generic search based on that information (e.g. counters reflecting the number of appearances of variables in conflicts) and to skip search nodes that are not related to the conflict, while backtracking. The latter case avoids the exploration of parts of the tree that are not related to the recent conflict; thus, those parts are infeasible.

Constraint propagation and conflict learning are very beneficial techniques for optimising a problem, because they can drastically reduce the search space that must be fully explored by the search. In addition, conflict-driven search is a generic search that performs well on average on hard problems.

1.3 Focus of this Thesis

In the previous section, a brief constraint-based solving scheme was given. In order to improve the solution process it is enough to improve an important part, e.g. building faster constraint propagators.

Boolean satisfiability (Sat) solvers are one form of heavily engineered constraint-based solver. They combine the advanced techniques of constraint propagation, conflict learning, and conflict-driven search with a very efficient implementation which can handle millions of constraints and hundreds of thousands of variables. Due to the fact that a problem must be encoded with Boolean variables and clauses, the limits in terms of a high-level model is much lower.

Recently, lazy clause generation (Lcg) has been proposed, which encodes a problem on a higher level, but lazily transforms the problem into a Sat problem. This allows an Lcg solver to use the advanced Sat technologies while efficiently solving problems that a Sat solver cannot handle directly.

The same approach underlies in modern satisfiability modulo theories (Smt) solvers. We develop the following techniques and solution approaches applicable to these solving approaches.

Unit two variables per inequality constraints: These constraints are one of the largest linear integer constraint classes that is solvable in polynomial time. They have the form $\pm x \pm y \leq d$ where x and y are integer variables and d an integer constant. Furthermore, they can express precedence constraints between activities in scheduling problems. Propagation algorithms determine the satisfaction of a set of these constraints and/or the implication of some constraints by a set of constraints.

A faster detection of satisfaction and implication will speed up the constraint propagation needed in a search node, thus reducing the overall runtime of the solution process.

We have developed new incremental satisfaction and implication algorithms that not only have an asymptotically better runtime complexity, but are also faster on sparse constraint systems. Moreover, we have developed algorithms for explaining unsatisfiability and implication that are specific for this class.

Cumulative resource constraints: These constraints appear in many scheduling problems modelling the relationship between scarce resources and activities that consume resources during their execution. Limited work has been done to explain their constraint propagation, and no explanations have been proposed for their constraint propagation in LCG solvers.

Explaining the propagation in the right way can improve the quality of nogoods and conflict-driven search. This improves solution processes for combinatorial problems that include these constraints.

Carpet cutting: It is a two-dimensional placement problem where carpet shapes must be cut from a carpet roll while minimising the wastage. We have developed a complete approach that takes account of all domain-specific constraints, uses cumulative constraints with explanations and improves the currently deployed method on industrial instances. An improvement not only cuts the cost for the company, but also decreases the required raw material.

1.4 Overview

In the next chapter, we first explain the basic concepts of constraint satisfaction problems and their solution, and introduce the basic terminology in this work. Then, we look at four different solving techniques, namely finite domain constraint propagation, Boolean satisfiability solving, lazy clause generation, and satisfiability modulo theories, which all are used for tackling combinatorial optimisation problems. We explain their similarities and their differences, and give the requirements on constraint propagators.

In the following, we give a brief summary of our contribution of each main chapter in this work.

Chapter 3 presents new incremental and non-incremental satisfaction and implication algorithms for unit two variable per inequality constraints which are

asymptotically faster than existing algorithms. These algorithms are based on a new theoretical result about reasoning in systems of these constraints. We enhanced these algorithms with explanations generating minimal unsatisfiable subsets and minimal implicants.

Chapter 4 deals with explanations of the propagation of cumulative resource constraints. We develop fine-grained strong explanations for the global cumulative constraint in a lazy clause generation solver, which are inspired by explanations generated by decomposition of these constraints.

Chapter 5 shows the power of cumulative propagation with explanation. We consider basic scheduling problems, namely the resource-constrained project scheduling problem, and its extension with generalised precedence relations. For these problems, we build a basic model and use a conflict driven search to minimise the project duration. On challenging standard benchmarks, our complete approach outperforms comparable state-of-the-art methods and closes many open problems.

Chapter 6 considers a carpet cutting application which can be viewed as a two-dimensional placement problem. In this problem carpet shapes must be cut from a carpet roll without overlapping. We develop a complete approach that minimises the carpet roll wastage while satisfying all domain-specific constraints. Then we show on industrial instances that our approach outperforms the current, incomplete approach.

Finally, we summarise this thesis and point to further research directions in Chap. 7.

2

Basic Principles

This chapter introduces the basic principles of constraint programming at first, then three specialised solving methods (finite domain propagation, Boolean satisfiability solver, and lazy clause generation) are explained in more detail. Finally, satisfiability modulo theory solvers, which have similarities to lazy clause generation, are described.

2.1 Constraint Programming

Constraint Programming (CP) (see e.g. Marriott and Stuckey 1998, Apt 2003) is one programming paradigm to solve combinatorial problems. Its roots come from Artificial Intelligence (AI) in the 70's. Early solution methods in AI can be understood as the generate-and-test principle: at first, generate a solution and then test if the solution meets all constraints. During that time, it was (widely) observed that this principle can lead to a "thrashing" behaviour (see e.g. Mackworth 1977, Gaschnig 1979). Thrashing, in this sense, means that an early "wrong" decision, that does not lead to a solution, is only detected when all possible remaining decisions are enumerated, which can be exponential in the size of the problem.

One idea to avoid thrashing is to test the feasibility of partial solutions and remove inconsistent values from variables' domains as the search progresses. This process is known in CP as *constraint propagation*, which is separately performed on each constraint by *constraint propagators*. Thus, the generate-and-test principle was reversed to a kind of test-and-generate principle.

Later, many techniques from operations research (OR) were adapted into CP. Mainly these techniques were deduction algorithms which are embedded as constraint propagators in CP.

In general, a CP problem is stated as constraint satisfaction problem. Its main solving principles are well-formulated by Baptiste *et al.* (2001) as follows.

- Strict separation of deductive methods (constraint propagators) and search algorithms
- Principle of locality (each constraint must be propagated as much as possible, independent of the presence or non-presence of other constraints)
- Strict separation between the (logical) representation of constraints and their propagation agreeing with the Kowalski equation for Logic Programming: Algorithm = Logic + Control (Kowalski 1979).

This section introduces some basic notations on constraint satisfaction problems and gives a basic understanding of the main components in the solution process: how each of them works and interacts with the other components. More details about some components are given in later sections in which solvers are described that solve particular constraint satisfaction problems.

2.1.1 Constraint Satisfaction Problem

In the following the basic notations for constraint programming are formally introduced. Given a sequence of variables $X = x_1, x_2, \ldots, x_n$ the domain, constraints, and then the constraint satisfaction problem are defined.

Definition 2.1 (Domain)**.** A *domain* D is a mapping from a sequence of variables X to a sequence of sets. A set $D(x_i)$ contains all elements that the variable x_i is allowed to take, and is called the domain of the variable x_i. The *initial domain* of a problem is denoted by D_{init}. A *false domain* maps at least one variable to an empty set.

When there is a total order of the elements of the domain $D(x)$ then the minimal and maximal element, respectively, are denoted by $\min_D(x)$ and $\max_D(x)$.

Definition 2.2 (Constraint)**.** A *constraint* $C(X)$ is an n-ary relation over a sequence of variables $X = x_1, x_2, \ldots, x_n$ and their respective domains $D(x_1)$, $D(x_2)$, $\ldots$, $D(x_n)$. Hence, a constraint describes the solution space by a subset of the Cartesian product over the domains $C(X) \subseteq D(x_1) \times D(x_2) \times \cdots \times D(x_n)$.

A constraint can be stated intentionally or extensionally.

Example 2.1. Some examples of constraints and their variables are given here:

- $C(x, y) = \{(1, \text{red}), (2, \text{blue}), (2, \text{green})\}$ with $x \in \{1, 2\}, y \in \{\text{blue}, \text{green}, \text{red}\}$,

- $C(x, y, z) = \{(x, y, z) \in \mathbb{N}^3 \mid x + 2y - z \leq 10\}$ with $x, y, z \in \mathbb{N}$,
- $C(b, x, y) = \{(b, x, y) \in \{true, false\} \times \mathbb{N}^2 \mid b \leftrightarrow x + 5 \leq y\}$ with $b \in \{true, false\}$, and $x, y \in \mathbb{N}$,
- $C(x, y) = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$ with $x, y \in \mathbb{R}$,

where $\mathbb{N}$ is the set of natural numbers and $\mathbb{R}$ is the set of real numbers. □

For brevity, constraints such as the second constraint in the previous example can be written as $x + 2y - z \leq 10$.

Definition 2.3 (Constraint Satisfaction Problem)**.** A *Constraint Satisfaction Problem* (Csp) consists of a sequence of variables $\mathcal{X} = x_1, x_2, \ldots, x_n$ with their respective domains $\mathcal{D}(x_1), \mathcal{D}(x_2), \ldots, \mathcal{D}(x_n)$, and a set of constraints $\mathcal{C}$ in which each constraint $C(X) \in \mathcal{C}$ is defined over a subsequence of variables $X \subseteq \mathcal{X}$. A Csp is denoted by the triple $(\mathcal{C}, \mathcal{X}, \mathcal{D})$.

Example 2.2. The Csp $(\{x + 2y + 3z \leq 10, x < y, y + z \leq 3\}, \mathcal{X} = x, y, z, \mathcal{D}(\mathcal{X}) = [\,0\,..\,10\,], [\,0\,..\,10\,], [\,0\,..\,10\,])$ involves three constraints on three variables where $[\,0\,..\,10\,]$ denotes the set of natural numbers from 0 to 10. □

Definition 2.4 (Assignment)**.** An *assignment* θ is a mapping from a sequence of variables $\mathcal{X} = x_1, x_2, \ldots, x_n$ to one element from each domain written as$\theta = \{x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots, x_n \mapsto d_n\}$ where $d_i \in D(x_i)$. The assignment θ *satisfies* a constraint $C(X)$ on a subsequence of variables $X \subseteq \mathcal{X}$ if $\theta(X) \in C(X)$ where $\theta(X) = \{\theta(x) \mid x \in X\}$.

Example 2.3. Assignments of the Csp in Ex. 2.2 are amongst others $\{x \mapsto 10, y \mapsto 0, z \mapsto 0\}$, $\{x \mapsto 1, y \mapsto 2, z \mapsto 0\}$, and $\{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$. The first assignment satisfies the first and the third constraints, the second assignment satisfies all constraints, and the third assignment the last two constraints. □

Definition 2.5 (Solution)**.** Given a Csp $\mathcal{P} = (\mathcal{C}, \mathcal{X}, \mathcal{D})$ and an assignment θ over the set of variables $\mathcal{X}$. The assignment θ is a *solution* of the problem if and only if it satisfies all constraints $C(X) \in \mathcal{C}$ where $X \subseteq \mathcal{X}$, *i.e.*,

$$\forall C(X) \in \mathcal{C} : \theta(X) \in C(X) \ .$$

The *solution space* (set of all solutions) of $\mathcal{P}$ is denoted by $Sol(\mathcal{P})$.

A Csp is *solvable* if and only if the solution space is not empty. In this work, we also write *feasible*, *satisfiable*, or *consistent* for solvable. If no solution exists then the Csp is *unsolvable*.

Example 2.4. Consider the CSP from Ex. 2.2 and the three assignments from Ex. 2.3. Just the second assignment is a solution for the CSP. For instance, other solutions are $\{x \mapsto 0, y \mapsto 3, z \mapsto 0\}$, $\{x \mapsto 0, y \mapsto 0, z \mapsto 3\}$, or $\{x \mapsto 1, y \mapsto 2, z \mapsto 1\}$. □

2.1.2 Solving

Solving of a CSP means finding a solution from the solution space or proving the infeasibility. In the last case the solution methods must be complete. In both cases, the process consists of propagation and search steps which are alternately applied until the answer is found. The propagation step tries to prune the problem while preserving all solutions, whereas the search step divides the problem into "simpler" subproblems and conquers them individually.

Before both steps are formally defined the notation of *strength* of domains and *equivalence* on CSPs are introduced.

Definition 2.6. Let D_1 and D_2 be domains over variables in X, then D_1 is called *stronger* than D_2 if $D_1(x) \subseteq D_2(x)$ for all $x \in X$. It is written as $D_1 \sqsubseteq D_2$.

Definition 2.7. Let $\mathcal{P}$ and $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_m$ be CSPs on the same variables. The CSP $\mathcal{P}$ is *equivalent* to the union of $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_m$ if $Sol(\mathcal{P}) = \bigcup_{i=1}^{m} Sol(\mathcal{Q}_i)$.

Propagation

The propagation step transforms a CSP to an equivalent CSP with a stronger domain. It is based on high-specialised propagators for each constraint or a set of constraints. These propagators take into account the local or structural information of the corresponding constraint in order to strengthen the domain.

Definition 2.8 (Propagator)**.** A *propagator* f is a monotonically decreasing function from domains to domains, such that $f(D_1) \sqsubseteq f(D_2)$ for all domains D_1 and D_2 with $D_1 \sqsubseteq D_2$, and preserves the solutions for any CSP $(\mathcal{C}, \mathcal{X}, \mathcal{D})$, *i.e.*, $Sol((\mathcal{C}, \mathcal{X}, \mathcal{D})) = Sol((\mathcal{C}, \mathcal{X}, f(\mathcal{D})))$. For ease of explanation, the notion of a propagator is lifted to CSPs, *i.e.*, $f((\mathcal{C}, \mathcal{X}, \mathcal{D})) = (\mathcal{C}, \mathcal{X}, f(\mathcal{D}))$.

In general, a propagator removes inconsistent elements from the variable's domains and can be defined in term of deduction rules.

Example 2.5. Consider the CSP with the sequence of variables $X = x, y, z$ from Ex. 2.2. A propagator f for the constraint $y + z \leq 3$ can be defined as follows.

$$f(D)(u) = \begin{cases} D(y) \cap (-\infty, 3 - \min_D(z)] & \text{if } u = y, \\ D(z) \cap (-\infty, 3 - \min_D(y)] & \text{if } u = z, \\ D(u) & \text{otherwise.} \end{cases}$$

It reduces the domain of y and z to $[\,0\,..\,3\,]$, and does not change the domain of other variables. □

Since a propagator only covers the propagation of one constraint, normally, the propagators are repeatedly applied until no more deduction is achieved, *i.e.*, a *fixpoint* is reached, or all values from one domain are removed. The resulting CSP is equivalent to the initial CSP before the application of the propagators. It contains the strongest domain with respect to the propagators and the domain in the initial CSP.

Definition 2.9 (Propagation Solver)**.** A *propagation solver propagate* on a CSP $\mathcal{P}$ repeatedly applies each propagator on the problem until no further change is made, *i.e.*, $propagate(\mathcal{P}') = \mathcal{P}'$, or one propagator deduces a false domain for at least one variable.

Example 2.6. Consider the CSP from Ex. 2.2, the propagator f from Ex. 2.5, the following propagator g for the constraint $x < y$

$$g(D)(u) = \begin{cases} D(x) \cap (-\infty, \max_D(y) - 1] & \text{if } u = x, \\ D(y) \cap [\min_D(x) + 1, \infty) & \text{if } u = y, \\ D(u) & \text{otherwise,} \end{cases}$$

and the propagator h for the constraint $x + 2y + 3z \leq 10$ with

$$h(D)(u) = \begin{cases} D(x) \cap (-\infty, 10 - 2y - 3z] & \text{if } u = x \text{ and } x,\ y,\ z \text{ are fixed,} \\ D(u) & \text{otherwise.} \end{cases}$$

A propagation solver with these propagators f, g, h transforms the CSP with the domain $[\,0\,..\,10\,]$ into an equivalent CSP with the domains $D(x) = [\,0\,..\,2\,]$, $D(y) = [\,1\,..\,3\,]$, and $D(z) = [\,0\,..\,2\,]$. □

Local Consistency A propagation solver is often characterised by its propagation strength, *i.e.*, the resulting CSP. This is measured by the *local consistency* the

propagation solver achieves on the domain with respect to the constraints.

The most well-known consistency is *arc consistency* (Mackworth 1977) for constraints with two variables. A binary constraint is arc consistent if and only if each element in the domain of one variable has a supporting element in the domain of the other variable with which it forms a solution of the constraint. It is equivalent for n-ary constraints, called *generalised arc consistency* (Mohr and Henderson 1986) or *domain consistency* (Hentenryck *et al.* 1998).

Definition 2.10 (Domain Consistency)**.** An n-ary constraint $C(X)$ is called *domain consistent* with respect to the domain D if and only if

$$\forall x_i \in X, \forall d_i \in D(x_i), \exists \theta \in D(X) : \\ \theta = \{x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots, x_i \mapsto d_i, \ldots, x_n \mapsto d_n\} \text{ satisfies } C(X) \ .$$

In general, propagators running in polynomial time can be built for linear inequalities, that enforces domain consistency on the propagated constraint.

Proposition 2.1. *Let $C(X) \equiv \sum_{i=1}^{n} a_i x_i \leq a_0$ be linear inequality constraint over the sequence of variables $X \equiv x_1, x_2, \ldots, x_n$, their respective coefficients $a_1, a_2, \ldots, a_n$, and the constant a_0. The following propagator f enforces domain consistency of $C(X)$ on $f(D)$ given any domain D over X.*

$$f(D)(u) = \begin{cases} D(x_i) \cap (-\infty, a_0 - b(i)] & \text{if } u = x_i \text{ and } i \in [\,1\,..\,n\,], \\ D(u) & \text{otherwise.} \end{cases}$$

where $b(i) = \sum_{j \in J(i)} \min\{a_j \cdot \min_D(x_j), a_j \cdot \max_D(x_j)\}$, and $J(i) = [\,1\,..\,n\,] \setminus \{i\}$.

These inequalities appear e.g. in scheduling problems involving cumulative resource constraints, in a special form where the variables x_i express if the activity i runs at a specific time period, and the constant a_0 represent the resource capacity. Later in this work, we deal with those cumulative resources.

For a propagation solver it is desirable that domain consistency on constraints is enforced, since it is the strongest form of local consistency that can be achieved, but it might be too expensive regarding runtime complexity. For instance, for some n-ary constraints it is NP-hard to decide whether they are feasible or infeasible with respect to the current domain. Consequently, a domain consistency check can take non-polynomial time and space for each element in the variable's domain, unless $P = NP$.

For this reason weaker forms of consistency were introduced, e.g. *bounds consistency* (see, e.g. Choi *et al.* 2006) for integer domains. The most used bounds

consistencies are *bounds(ℤ)* and *bounds(ℝ)* consistency. Instead of looking for for supports for each element in the domain only a support is sought for the minimum and the maximum.

Definition 2.11 (Bounds(ℤ) Consistency)**.** An n-ary constraint $C(X)$ is *bounds(ℤ) consistent* with respect to the domain D if and only if

$$\begin{aligned}\forall x_i \in X, \forall d_i \in \{\min_D(x_i), \max_D(x_i)\}, \forall x_j \in X \setminus \{x_i\},\\ \exists d_j \in \mathbb{Z} \text{ with } \min_D(x_j) \leq d_j \leq \max_D(x_j) :\\ \{x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots, x_n \mapsto d_n\} \text{ satisfies } C(X) \ .\end{aligned}$$

Example 2.7. Consider the constraint $\mathsf{odd}(x)$ on the variable x. This constraint is satisfied if x is assigned an odd number. The constraint is bounds(ℤ) consistent with respect to the domain $D(x) = [\,1\,..\,5\,]$, but not domain consistent. □

The bounds(ℝ) consistency relaxes the bounds(ℤ) consistency in such a way that the supporting elements can be real numbers.

Definition 2.12 (Bounds(ℝ) Consistency)**.** An n-ary constraint $C(X)$ is *bounds(ℝ) consistent* with respect to the domain D if and only if

$$\begin{aligned}\forall x_i \in X, \forall d_i \in \{\min_D(x_i), \max_D(x_i)\}, \forall x_j \in X \setminus \{x_i\},\\ \exists d_j \in \mathbb{R} \text{ with } \min_D(x_j) \leq d_j \leq \max_D(x_j) :\\ \{x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots, x_n \mapsto d_n\} \text{ satisfies } C^{\mathbb{R}}(X) \ ,\end{aligned}$$

where $C^{\mathbb{R}}(X)$ is the real relaxation of $C(X)$.

Example 2.8. Consider the constraint $3x+2y = z$ with the domains $D(x) = [\,0\,..\,2\,]$, $D(y) = [\,0\,..\,2\,]$, and $D(z) = [\,1\,..\,9\,]$. The domain D is bounds(ℝ) consistent, but not bounds(ℤ) consistent. □

To enforce bounds(ℤ) consistency becomes prohibitive for some constraints, due to the high runtime cost. For instance, a bounds(ℤ) consistency check is *NP*-complete for linear equality constraints, while a linear bounds(ℝ) consistency check exists (see e.g. Schulte and Stuckey 2005).

Propagation Sequence In general, the propagation solver operates with a priority queue to which propagators are added for their execution. Propagators are only queued in two situations: if their corresponding constraint is added to the Csp or the domain of at least one of its variables was changed by another propagator.

Algorithm 2.1: *solve*($\mathcal{P}$) — Depth-first search with chronological backtracking

Input: Csp $\mathcal{P} = (\mathcal{C}, \mathcal{X}, \mathcal{D})$
Output: *true*, if $\mathcal{P}$ has a solution; otherwise *false*

```
1 P' := propagate(P);
2 if ∃x ∈ X : D'(x) = ∅ then return false;
3 if ∃x ∈ X : |D'(x)| > 1 then
4     Q_1, Q_2, ..., Q_k := divide(P');
5     for i = 1, 2, ..., k do
6         if solve(Q_i) = true then return true;
7     return false;
8 return true;
```

Normally, propagators with low runtime complexity are given a higher priority than others. For instance, a binary constraint usually has the highest priority, whereas an n-ary constraint has a low priority. In this way, the solver should be more efficient than a solver that applies all propagators in a loop until the fixpoint is reached.

Search and Backtracking

While propagation transforms a Csp to one equivalent Csp, search splits a Csp into two or more Csps, the union of which is equivalent to the initial Csp. Each split Csp is solved individually. Hence, the search follows the "divide-and-conquer" principle with an additional propagation step on each Csp.

In general, complete solvers use a tree search algorithm in order to solve these problems. A node is associated with a propagation step and decision step. Algorithm 2.1 shows an outline of a basic complete solving algorithm. This (recursive) algorithm *solve*($\mathcal{P}$) traverses the search tree in a *depth-first* manner with *chronological backtracking*. First, it propagates $\mathcal{P}$ in order to strengthen the domain (line 1). If the propagated Csp $\mathcal{P}'$ contains an empty domain of at least one variable, *i.e.*, the local $\mathcal{P}$ is unsolvable, then this instance of *solve*($\mathcal{P}$) terminates and return *false* (line 2). In the other case, the algorithm continues its search (lines 4 to 7) if not all variables are fixed (line 3). Otherwise, it has proven the feasibility of $\mathcal{P}'$, consequently of the initial Csp as well, and terminates with *true* (line 8). If an unfixed variable in the problem exists then the decision level is incremented and the procedure *divide* (line 4) splits the problem into k subproblems $\mathcal{Q}_1$, $\mathcal{Q}_2$, ..., $\mathcal{Q}_k$. After this, the algorithm iterates over all subproblems (lines 5 and 6). The algorithm stops at the first subproblem that is proven to be feasible and returns *true* (line 6). If all subproblems are infeasible then the algorithm terminates with *false* (line 7).

The procedure *divide* (line 4) is of particular interest. It splits the CSP into several problems by adding constraints to the CSP. The union of these problem are equivalent to the original CSP. Normally, these constraints partition the search space in which each part corresponds to one subproblem. For instance, suppose the CSP $(\mathcal{C}, \mathcal{X}, \mathcal{D})$ includes the variable x with a domain $D_x = [\,0\,..\,3\,]$ then the problem can be equivalently split into $(\mathcal{C} \cup \{x = 0\}, \mathcal{X}, \mathcal{D})$, $(\mathcal{C} \cup \{x = 1\}, \mathcal{X}, \mathcal{D})$, ..., $(\mathcal{C} \cup \{x = 3\}, \mathcal{X}, \mathcal{D})$ where the added constraints $x = 0$, $x = 1$, ..., $x = 3$ partition the search space.

Due to the NP-hardness of CSPs, a solving algorithm running in polynomial time and space for all inputs does not exist, unless P = NP. Consequently, generic (problem independent) or domain-specific heuristics are used for the *divide* procedure which drives the search and shapes the search tree. Their aims are to minimise the size of the search tree, *i.e.*, to make the important decisions high up in the search tree, and to order the search tree in such a way that a solution is situated far left in the search tree. If a heuristic can achieve these both targets then it can quickly find a solution or prove the infeasibility of an instance.

Many heuristics consist of a *variable selection* and a *branching* function. The variable selection function chooses an unfixed variable by some criteria and the branching decides how to partition the domain of a variable and creates the corresponding subproblems. A good selection function is for example *first fail* (Haralick and Elliott 1980) which chooses the variable with the smallest domain size.

Example 2.9. This example shows a model and the first steps of solving process for an one-machine scheduling problem. we will use it as a running example of solving approaches discussed in this chapter.

Suppose an one-machine scheduling problem with four tasks A, B, C, and D is given, where these tasks have to be executed exclusively on one machine and the execution of a task cannot be interrupted. This means two tasks cannot be run in the same time period. The attributes of the tasks A, B, C, and D respectively are their processing times 5, 3, 2, and 3, their earliest start time 0, 1, 1, 6, and their latest completion time 14, 12, 12, and 12. The goal is to find a solution that assigns integer values to the start time variable s_A, s_B, s_C, and s_D, so that no task is run concurrently, all tasks start not before their earliest start time, and end not later than their latest completion time.

This problem is modelled as a CSP $(\mathcal{C},\ \{b_{AB}, b_{AC}, b_{AD}, b_{BC}, b_{BD}, b_{CD}, s_A, s_B, s_C, s_D\},\ \mathcal{D})$ where $\mathcal{D}(b_{ij}) = \{\mathit{true}, \mathit{false}\}$ $(\forall i, j \in \{A, B, C, D\})$, $\mathcal{D}(s_A) = [\,0\,..\,9\,]$,

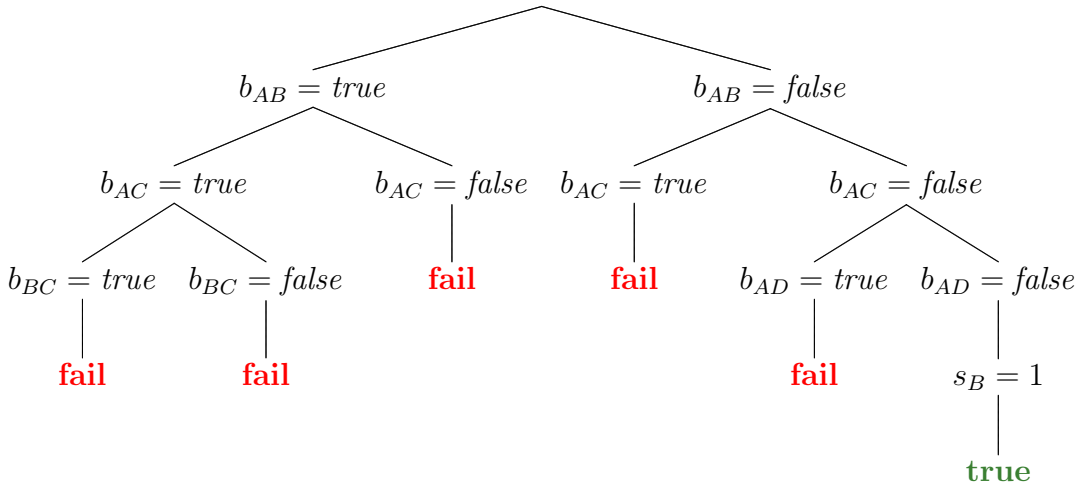


Figure 2.1: The corresponding search tree for Ex. 2.9.

$\mathcal{D}(s_B) = [1\,..\,9]$, $\mathcal{D}(s_B) = [1\,..\,10]$, $\mathcal{D}(s_B) = [6\,..\,9]$, and the set of constraints

$$\begin{aligned}
\mathcal{C} = \{ \quad & C_1 \equiv b_{AB} \leftrightarrow s_A + 5 \leq s_B \ , & & C_2 \equiv \neg b_{AB} \leftrightarrow s_B + 3 \leq s_A \ , \\
& C_3 \equiv b_{AC} \leftrightarrow s_A + 5 \leq s_C \ , & & C_4 \equiv \neg b_{AC} \leftrightarrow s_C + 2 \leq s_A \ , \\
& C_5 \equiv b_{AD} \leftrightarrow s_A + 5 \leq s_D \ , & & C_6 \equiv \neg b_{AD} \leftrightarrow s_D + 3 \leq s_A \ , \\
& C_7 \equiv b_{BC} \leftrightarrow s_B + 3 \leq s_C \ , & & C_8 \equiv \neg b_{BC} \leftrightarrow s_C + 2 \leq s_B \ , \\
& C_9 \equiv b_{BD} \leftrightarrow s_B + 3 \leq s_D \ , & & C_{10} \equiv \neg b_{BD} \leftrightarrow s_D + 3 \leq s_B \ , \\
& C_{11} \equiv b_{CD} \leftrightarrow s_C + 2 \leq s_D \ , & & C_{12} \equiv \neg b_{CD} \leftrightarrow s_D + 3 \leq s_C \qquad \} \ .
\end{aligned}$$

The Boolean variables b_{ij} describe whether the task i runs before or after the task j. The propagator f for a constraint of the form $b \leftrightarrow x + d \leq y$ is defined as follows:

$$f(D)(u) = \begin{cases}
D(b) \cap \{true\} & \text{if } u = b \text{ and } \max_D(x) + d \leq \min_D(y), \\
D(b) \cap \{false\} & \text{if } u = b \text{ and } \min_D(x) + d > \max_D(y), \\
D(x) \cap (-\infty, \max_D(y) - d] & \text{if } u = x \text{ and } b = true, \\
D(x) \cap [\min_D(y) - d + 1, \infty) & \text{if } u = x \text{ and } b = false, \\
D(y) \cap [\min_D(x), \infty) & \text{if } u = y \text{ and } b = true, \\
D(y) \cap (-\infty, \max_D(x) + d - 1] & \text{if } u = y \text{ and } b = false, \\
D(u) & \text{otherwise.}
\end{cases}$$

The problem is solved by Alg. 2.1 where the *divide* procedure works as follows: in each node an unfixed variable is selected in the order stated above, and its current domain is branched by enumeration of the elements starting with the *true* element for Boolean variables and the smallest element for integer variables. The first few

steps of the algorithm are outlined in the following, and the search tree is shown in Fig. 2.1.

1. b_{AB} is assigned *true*.
   → Propagation on C_1: $f(D)(s_B) = [\,5\,..\,9\,]$ and $f(D)(s_A) = [\,0\,..\,4\,]$.
   → Propagation on C_6: $f(D)(b_{AD}) = \{true\}$.
2. b_{AC} is assigned *true*.
   → Propagation on C_3: $f(D)(s_C) = [\,5\,..\,10\,]$.
3. b_{BC} is assigned *true*.
   → Propagation on C_7: $f(D)(s_C) = [\,8\,..\,10\,]$ and$f(D)(s_B) = [\,5\,..\,7\,]$.
   → Propagation on C_{11}: $f(D)(b_{CD}) = \{false\}$.
   → Propagation on C_{12}: $f(D)(s_D) = [\,6\,..\,7\,]$.
   → Propagation on C_{10}: $f(D)(b_{BD}) = \{true\}$.
   → Propagation on C_9: $f(D)(s_D) = \emptyset$.
   → Backtracking to the last decision and choosing the next subproblem.
3. b_{BC} assigned *false*. Propagation leads to an empty domain. Backtracking.
3. No alternative element for b_{BC}. Backtracking.
2. b_{AC} assigned *false*. Propagation leads to an empty domain. Backtracking.

After five failures and eleven explored nodes Alg. 2.1 proves the satisfiability of the instance. The generated solution leads to a schedule in that the tasks A, B, C, and D are started at time period 9, 1, 4, and 6 respectively. □

Backtracking An issue with the basic algorithm is the chronological backtracking (Gaschnig 1979). If a subproblem was proven to be infeasible then the search backtracks to the previous decision level and selects another unexplored subproblem. But it might be that the previous decision is not related to the conflict. Hence, all subproblems also are infeasible. Instead of proving the infeasibility of the remaining unexplored subproblems (which is what chronological backtracking does) the search can backtrack further. Different *non-chronological backtracking* strategies have been proposed to overcome that kind of thrashing; e.g. backjumping (Gaschnig 1979).

Nogoods and Explanations Normally, the search tree includes redundancies in different subtrees including infeasible solution states. In order to avoid a repeated proof of the same or similar infeasibility (in a different subtree) some search algorithms analyse conflicts and, based on that, they generate constraints which are

called *nogoods*. These constraints are redundant and are permanently or temporarily added to the initial Csp, hence they also appear in the subproblems that are created during the search. They are inferred from explanations and/or search decisions where an *explanation* records the reason for value removals during propagation. Explanations can also be used to short-circuit propagation.

Beside the pruning of the search space the nogoods can be used to guide a conflict-driven search. For instance, if a variable often appears in nogoods then it might be better to branch over it in higher level in the search tree, because it may narrow the search space more than other variables. Another advantage of nogoods is that they can reformulate the chronological backtracking. Instead of only backtracking to the previous decision level, the search can backtrack to the level at which the nogood is for the first time no longer infeasible.

Restarts A restart abandons the current search and starts the search from the beginning. The idea is to explore another part of the search tree which may lead to faster solving of the problem and to escape hard-to-prove infeasible subtrees. A restart only makes sense if the search from the initial Csp will change, e.g. nogoods were added, and/or the search decisions depend on dynamic information which is gathered during the search and propagation, e.g. impact-based searches (Refalo 2004), or are randomised partly or fully.

A problem with restart occurs if a Csp is infeasible. In this case the search must exhaustively explore the search space for the infeasible proof which then can be stopped by a restart. Because of that, restart policies increase the time between restarts in the long run. However, a restart can be either advantageous or disadvantageous in this case.

Summary Solving a Csp is a complex task that consists of carefully modelling the problem, *i.e.*, choosing the right constraints, and carefully selecting the search heuristic. The aim is to choose those constraints with a positive trade-off between pruning the search space most and time to be propagated, and to select a heuristic that minimises the size of the search space. Since Csps typically are NP-hard, searches can involve a substantial number of backtracking steps and many redundancies in the search tree. For both of these reasons, non-chronological backtracking strategies, nogood generation, and restart policies can be employed to make a search algorithm more robust.

2.1.3 Optimisation

In comparison to the satisfaction problem, an optimisation problem does not demand just any solution, but the "best" one. The quality of a solution is measured with a given objective function *obj* that assigns an integer number to each assignment. Here, we concentrate on minimisation problems, since maximisation problems can be transformed to minimisation problems by using $-obj$ as objective function.

Definition 2.13 (Minimisation Problem)**.** A *minimisation problem* is denoted by a tuple $(\mathcal{P}, obj)$ where $\mathcal{P}$ is a CSP and *obj* is an *objective function* or, in short, *objective* that is an expression over the variables in $\mathcal{P}$ and that assigns an integer number to each assignment.

The goal is to find a solutions $\theta \in Sol(\mathcal{P})$ for that holds $obj(\theta) \leq obj(\theta')$ for all $\theta' \in Sol(\mathcal{P})$. Such a solution θ is called *optimal* or *minimal*.

Example 2.10. Consider the one-machine problem from Ex. 2.9 and the objective function $obj = s_B + 2 \cdot s_C$. This problem has two solutions $\{s_A \mapsto 9, s_B \mapsto 1, s_C \mapsto 4, s_D \mapsto 6\}$ and $\{s_A \mapsto 9, s_B \mapsto 3, s_C \mapsto 1, s_D \mapsto 6\}$, with their respective objective values 9 and 5. The second solution is the minimal one. □

Two main techniques used for solving minimisation problems in CP are dichotomic search and branch-and-bound (Land and Doig 1960). Both techniques transform the initial minimisation problem into a sequence of CSPs for which they continuously narrow the solution space by adding constraints on the objective value. Note that the objective is modelled as one or more constraints in each CSP.

The basic dichotomic search acts as a binary search on the initial range of the objective *obj*: in each step, it transforms the minimisation problem into a CSP by adding constraints for the lower and upper bound on the objective. First, it splits the current range $[\,LB\,..\,UB\,]$ of the objective into two parts $[\,LB\,..\,p\,]$ and $(p, UB]$ where p is the midpoint $(UB - LB)/2$. Second, it solves the CSP with the first range on the objective, and then it continues with the second range if the CSP was proven infeasible. These steps are applied recursively until the lower and upper bound of the objective match each other. Hence, at most $\mathcal{O}(\log(UB - LB))$ CSP are solved. Note that an initial (trivial) bound can be assumed on the objective.

Example 2.11. Consider the minimisation problem from Ex. 2.10. Given a trivial lower bound of 0 and an upper bound of 9 from the first found solution the basic dichotomic search would produce this sequence of ranges for the objective: $[\,0\,..\,4\,]$, $[\,5\,..\,7\,]$, $[\,5\,..\,6\,]$, $[\,5\,..\,5\,]$. □

The advantage of this search is that CSP solvers (that support all constraints of the problem) can solve the minimisation problem. Beside the described basic search, many other variants have been developed (see e.g. Sellmann and Kadioglu 2008). Some of them are problem specific.

Instead of solving a sequence of CSPs in individual search trees the branch-and-bound algorithm (Land and Doig 1960) solves them in one search tree by tracking the best found objective value *best* so far and enforcing that each new solution must be better than the current best. During the search the algorithm (continuously) checks if the constraint $obj < best$ is satisfied and updates the best value *best* once a new solution is found. Note that the best value is a global value that is not relaxed during backtracking, and a full traversal of the search tree is necessary.

Example 2.12. Consider the minimisation problem from Ex. 2.10. A trivial lower bound of 0 is given, but no upper bound. The search finds the first solution as described in Ex. 2.9 with the objective value 9. Hence, the upper bound is updated to 9. Then it backtracks to the previous decision and explores the next subproblem in which s_B is assigned 2. The following propagation leads to an empty domain resulting in backtracking. Then the subproblem in which s_B is fixed to 3 is explored. The following propagation fixes the remaining unfixed variable s_C to 1 leading to the new upper bound 5 on the objective value. Backtracking is initiated and the search terminates with the optimal value 5, since all subproblems were explored for the previous decision levels. □

2.2 Finite Domain Propagation

Finite domain propagation (see, e.g. Schulte and Stuckey 2008) is the branch of constraint programming in which the domain of a CSP is finite. It is a powerful generic approach for tackling combinatorial (optimisation) problems due to its high-level expressiveness (of constraints) and highly-specialised propagators. Because of the finite domains the domains are normally encoded with integers which in turn can be an abstraction of any entity as e.g. a colour, geometrical object etc.

Definition 2.14 (Finite Domain Problem)**.** A CSP $\mathcal{P} = (\mathcal{C}, \mathcal{X}, \mathcal{D})$ is called a *finite domain* (FD) problem if and only if $\mathcal{D}(x) \subseteq \mathbb{Z}$ and $\mathcal{D}(x)$ is finite for each $x \in \mathcal{X}$.

The domains of variables can be encoded in different ways. The most common ones are a set of integers, a range of integers, and a set of ranges of integers (this corresponds to a range of integers with "holes" in the domain). Note that a Boolean variable can be encoded by the integers 0 and 1 which respectively represent the

false and *true* value. In this work, the domains of variables are represented as a range of integers that is denoted by $[\,l\,..\,u\,] = \{i \in \mathbb{Z} \mid l \leq i \leq u\}$, and holes are not allowed. Most FD solvers provide this representation.

Example 2.13. The one-machine problem (see Ex. 2.9 on page 15) is an FD problem. We consider Boolean variables to be integer variables with domains $\{0, 1\}$ where 0 and 1 represent *false* and *true* respectively. □

2.2.1 Common Constraints

In general, an FD solver provides a large variety of different constraints that make it easy for an user not only to model a problem, but also to modify the model.

Atomic constraints Atomic constraints are unary constraints, *i.e.*, the constraint involves one variable, that restricts the bounds on the variables or the values that a variable can or cannot take.

Definition 2.15 (Atomic constraints)**.** A constraint $C(x)$ on the variable x is called *atomic* if it has one of the following forms: $x \leq d$, $x \geq d$, $x = d$, or $x \neq d$, where d is an integer.

The first inequality in the definition refers to an upper bound restriction on the variable and the second one to a lower bound restriction. The inequalities $x < d$ and $x > d$ equate to the respective atomic constraints $x \leq d - 1$ and $x \geq d + 1$.

The first three constraints only have to be propagated once. If the domains are encoded as ranges of integers then the last constraint ($x \neq d$) can be propagated if and only if one bound on the variable equals d. Note that the disequality only has to propagate once, too, if the domain is represented as a range of integers with holes.

Arithmetic Constraints These constraints describe (amongst other things) inequalities ($<$, $>$, $\leq$, or $\geq$), equalities ($=$), and disequalities ($\neq$) between at least two variables. Propagators for these constraints are mainly bounds($\mathbb{R}$) consistent.

In this work, we use a special case of linear inequalities of the form $\sum_{i \in [\,1\,..\,n\,]} a_i x_i \leq d$ where $d \in \mathbb{Z}$, $a_i \in \mathbb{N}$ and x_i a Boolean variable for all $i \in [\,1\,..\,n\,]$, and unit-two-variables per inequality constraints, *i.e.*, $ax+by \leq d$ where $d \in \mathbb{Z}$, $\{a, b\} \subseteq \{-1, 0, 1\}$ and x, y are integer variables. An important specialisation of the second constraint is the difference constraint (also called separation constraint) in which $a = 1$ and $b = -1$. For instance, this constraint models the precedence relation between two activities in scheduling problems.

Global Constraints Global constraints are complex n-ary constraints and play an important role for CP systems in general, since they are a "high-level" abstraction for complex relations between n variables. They may have a decomposed version made up of simpler constraints, but usually direct propagators for global constraints can exploit the structural information in order to prune the domains of variables more. Furthermore, a decomposition can be prohibitive if too many simple constraints are introduced or the propagation of these constraints leads to worse queueing behaviour during the propagation solver. An online catalogue of global constraints (Beldiceanu *et al.* 2007a) can be found at `http://www.emn.fr/z-info/sdemasse/gccat/`.

Probably, the most well-known global constraint is `alldifferent` (Régin 1994) which expresses the restriction that all variables must be assigned a different value. Its decomposed version is made of $(n-1)n/2$ disequalities $x_i \neq x_j$ for all $i, j \in [\,1\,..\,n\,]$ with $i \neq j$ where n is the number of variables. For instance, the (standard) Sudoku puzzle consists of a 9×9 grid in which each field must be filled in with a number in $[\,1\,..\,9\,]$. But each field in each row and column must be filled in with a different value than the other fields in the row or column. Moreover, the grid is partitioned into nine 3×3-squares in which each field must also take a different value than the other fields in the square. Hence, an `alldifferent` constraint is used for each row, each column, and each 3×3 square.

Example 2.14. The classical example to show that a global propagator can yield a stronger propagation than propagators of the decomposed version of the constraint is as follows. Let x, y, and z be variables that must take different values from the domain $\{1, 2\}$. Clearly, this problem is infeasible, since three variables must share only two values. A global `alldifferent` propagator immediately detects the infeasibility, but not the decomposed version consisting of the primitive constraints $x \neq y$, $y \neq z$, $z \neq x$. □

Another important global constraint is the cumulative resource constraint (Aggoun and Beldiceanu 1993), especially in areas (amongst others) such as scheduling, packing and cutting. We denote this constraint by `cumulative`. The constraint models the relationship between scarce resources and activities that require some part of the resource capacity for their execution. It ensures a non-overload of the resource capacity at any time period during the planning horizon. Since this constraint is one focus point in this work, a detailed description is given in Chap. 4.

Normally, different propagators exists for a global constraint. They differ in their propagation strength and their runtime cost. The choice of which one or ones to

run depends whether their runtime costs can be compensated by their reductions of the search space. This can be different from problem to problem.

Reified Constraints A reified constraint for a constraint $C(X)$ relates its truth value to a Boolean variable b by a (logical) equivalence relation, written as $b \leftrightarrow C(X)$. Reified constraints allow logical relations to be expressed between different constraints rather than simply conjunction of constraints.

Example 2.15. Consider two activities 1 and 2 with their start times s_1 and s_2 and their durations d_1 and d_2. The activities are not allowed to overlap at any time, *i.e.*, one activity must finish before the other can start. The disjunction $s_1 + d_1 \leq s_2 \vee s_2 + d_2 \leq s_1$ reflects the non-overlap constraint. It can be encoded with reification as $b_1 \leftrightarrow s_1 + d_1 \leq s_2$, $b_2 \leftrightarrow s_2 + d_2 \leq s_1$, and $b_1 \vee b_2$. Those reified constraints we already use in the running example (see Ex. 2.9). □

2.3 Boolean Satisfiability Solving

Boolean satisfiability (SAT) solving is a well-understood constraint solving technique. It has experienced a drastic improvement in the last two decades. Nowadays, modern SAT solvers (e.g. Moskewicz *et al.* 2001) can often handle problems with millions of constraints and hundreds of thousands of variables. But still many problems are difficult to encode into SAT without breaking these implicit limits.

The SAT problems are CSPs that are made of Boolean variables, *i.e.*, the domain of variables is a subset of $\{true, false\}$, and constraints are Boolean formulas. Normally, these formulas are given in *conjunctive normal form* (CNF) where a CNF is a conjunction of clauses $(C_1 \wedge C_2 \wedge \cdots \wedge C_m)$. A *clause* is a disjunction of literals $(l_1 \vee l_2 \vee \cdots \vee l_n)$ where a *literal* l is a Boolean variable b or its negation $\neg b$. A CNF can be constructed for any Boolean formula by using a linear encoding (Tseitin 1968).

Definition 2.16 (Boolean Satisfiability Problem)**.** A CSP $\mathcal{P} = (\mathcal{C}, \mathcal{B}, \mathcal{D})$ is called a *Boolean satisfiability* (SAT) problem if and only if all constraints $C \in \mathcal{C}$ are clauses and each variable $b \in \mathcal{B}$ is a Boolean variable, *i.e.*, $\mathcal{D}(b) \subseteq \{true, false\}$. The set of constraints is also called *clause database*.

The following SAT problem is a running example throughout this subsection.

Example 2.16. Consider the one-machine problem from Ex. 2.9 on page 15 with the set of four tasks $\mathcal{V} = \{A, B, C, D\}$ and their processing times 5, 3, 2, 3, their release dates $0, 1, 1, 6$, and their deadlines $14, 12, 12, 12$, respectively.

Here, this CSP is transformed to a SAT problem $(\mathcal{C}, \mathcal{B}, \mathcal{D})$ in which we split the set of variables and the set of constraints into two parts: $\mathcal{B} = \mathcal{B}_1 \cup \mathcal{B}_2$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$. The Boolean variables in $\mathcal{B}_1$ represent the precedence relation between each pair of tasks and the Boolean variables in $\mathcal{B}_2$ are a possible encoding for the start times for each task. A start time t of task i is reflected by an assignment of $false$ to b_i^{t-1} and *true* to b_i^t.

The clauses in $\mathcal{C}_1$ describe the relationship between the start times of a task and the clauses in $\mathcal{C}_2$ reflect all possible propagation on the start times with respect to precedence relation between tasks.

These sets are defined as follows:

$$
\begin{aligned}
\mathcal{B}_1 &= \{b_{ij} \mid \forall i, j \in \mathcal{V}, i \neq j : \text{ task } i \text{ runs before task } j\} \\
\mathcal{B}_2 &= \{b_i^t \mid \forall i \in \mathcal{V}, \forall t \in [\, est_i \,..\, lct_i - p_i \,] : \text{ task } i \text{ starts before or at time period } t\} \\
\mathcal{C}_1 &= \{b_i^t \rightarrow b_i^{t+1} \mid \forall i \in \mathcal{V}, \forall t \in [\, est_i \,..\, lct_i - p_i - 1 \,]\} \\
\mathcal{C}_2 &= \{b_{ij} \wedge \neg b_i^{t-1} \rightarrow \neg b_j^{p_i+t-1} \mid \forall i, j \in \mathcal{V}, \forall t \in [\, est_i \,..\, lct_i - p_i \,]\} \\
&\quad \cup \{b_{ij} \wedge b_j^t \rightarrow b_i^{t-p_i} \mid \forall i, j \in \mathcal{V}, \forall t \in [\, est_j \,..\, lct_j - p_j \,]\}
\end{aligned}
$$

where s_i, est_i, lct_i, and p_i are the (possible) start time, the earliest start time, the latest completion time, and the processing time of i, respectively. The Boolean variables b_{ij} with $i < j$ are names for $\neg b_{ji}$, *i.e.*, $b_{ij} \equiv \neg b_{ji}$. □

2.3.1 Solving

The SAT propagation is mainly based on resolution, unit propagation (a special case of resolution), and pure literal elimination which is mainly unused in modern SAT solvers. The *resolution* step looks for particular pairs of clauses that have a literal l in common, but in one clause it appears in the negated form $\neg l$. Then it deduces a new clause (called *resolvent*) consisting of the remaining literals of both clauses.

$$\frac{(l \vee U) \wedge (\neg l \vee V)}{U \vee V} \tag{2.1}$$

where U and V are clauses. If $U \vee V$ is empty, *i.e.*, does not contain any literal, then infeasibility of the problem is proved. The *unit propagation* propagates clauses in which only one undecided literal is left and all other literals are *false*. Those clauses are called *unit*. The propagation can be written as the following rule:

$$\frac{(l \vee l_1 \vee \cdots \vee l_k) \wedge \neg l_1 \wedge \cdots \wedge \neg l_k}{l} \ .$$

Algorithm 2.2: *dpll_solve*($\mathcal{P}$) — DPLL algorithm with chronological backtracking

Input: SAT $\mathcal{P} = (\mathcal{C}, \mathcal{B}, \mathcal{D})$
Output: *true*, if $\mathcal{P}$ has a solution; otherwise *false*

```
1 P' := unit_propagate(P);
2 P' := pure_literal_elimination(P');
3 if ∃x ∈ X' : D'(x) = ∅ then return false;
4 if ∀b ∈ B' : |D'(b)| = 1 then return true;
5 l := propose_literal(P');
6 return dpll_solve(P' + l) or dpll_solve(P' + ¬l);
```

The *pure literal elimination* looks for literals, that only appear either in the positive or in the negative form in all not (yet) satisfied clauses, and assigns *true* to them.

$$\frac{\forall C \in \mathcal{C} : \neg l \notin C}{l} \ ,$$

where $\mathcal{C}$ is a set of clauses that are not satisfied yet. Note that the rule is not a necessary condition for the feasibility of a SAT problem, *i.e.*, there can exist a solution in which the literal l is assigned *false*.

Typically, complete SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm (Davis and Putnam 1960, Davis *et al.* 1962). An outline of this algorithm is given in Alg. 2.2, which is a special case of the solving scheme *solve* on page 14, although the algorithm only applies unit propagation and pure literal elimination. The procedure *propose_literal* returns an undecided literal over which the search is branched. Different heuristics have been proposed for this procedure.

Example 2.17. Consider the SAT problem from Ex. 2.16. Now, we solve the problem by the DPLL algorithm (Alg. 2.2), but without the pure literal elimination. The procedure *propose_literal* only proposes unfixed literals in the following order b_{AB}, b_{AC}, b_{AD}, b_{BC}, b_{BD}, b_{CD}, b_A^0, b_A^1, ..., b_B^1, b_B^2, ..., b_C^1, b_C^2, ..., b_D^6, b_D^7, ... At first the search branches over $b_i = \mathit{true}$ and then $b_i = \mathit{false}$ after backtracking.

This search is the same search for the precedence Boolean variables as in Ex. 2.9 for solving the same problem, but modelled as a CSP. It is slightly different in the start times of tasks.

However, the DPLL algorithm with this search spans the same search tree as in Ex. 2.9 for proving the satisfiability. The first steps of the algorithm are as follows in which the propagation is only shown for constraints in $\mathcal{C}_2$ (see Ex. 2.16):

1. b_{AB} is assigned *true*.

   → Unit propagation on $\neg b_{AB} \vee \neg b_B^4$: $s_B^4 = \mathit{false}$, *i.e.*, $s_B \geq 5$.

→ Unit propagation on $\neg b_{AB} \vee b_A^4$: $b_A^4 = \mathit{true}$, *i.e.*, $s_A \leq 4$.

→ Unit propagation on $b_{AD} \vee \neg b_A^4$: $b_{AD} = \mathit{true}$.

2. b_{AC} is assigned *true*.

→ Unit propagation on $\neg b_{AB} \vee \neg b_C^4$: $b_C^4 = \mathit{false}$, *i.e.*, $s_C \geq 5$.

3. b_{BC} is assigned *true*.

→ Unit propagation on $\neg b_{BC} \vee b_B^4 \vee \neg b_C^7$: $b_C^7 = \mathit{false}$, *i.e.*, $s_C \geq 8$.

→ Unit propagation on $\neg b_{BC} \vee b_B^7$: $b_B^7 = \mathit{true}$, *i.e.*, $s_B \leq 7$.

→ Unit propagation on $\neg b_{CD} \vee b_C^7$: $b_{CD} = \mathit{false}$.

→ Unit propagation on $b_{CD} \vee b_D^7$: $b_D^7 = \mathit{true}$, *i.e.*, $s_D \leq 7$.

→ Unit propagation on $b_{BD} \vee \neg b_B^7$: $b_{BD} = \mathit{true}$.

→ Conflict clause $\neg b_{BD} \vee b_B^4 \vee \neg b_D^7$.

→ Backtracking to the last decision and choosing the next subproblem.

3. b_{BC} assigned *false*. Propagation leads to a conflict. Backtracking.

3. No alternative element for b_{BC}. Backtracking.

2. b_{AC} assigned *false*. Propagation leads to a conflict. Backtracking. □

Modern Sat solvers enhance the Dpll algorithm with *conflict learning* (*conflict analysis*), non-chronological backtracking, conflict-driven search and restart policies. The conflict analysis starts once unit propagation hits an unsatisfiable clause. It infers a new clause (a nogood) from the clauses and/or decision that lead to the infeasibility, and adds the nogood permanently or temporarily to the set of clauses. Then it backjumps to that decision level in which the nogood is unit, *i.e.*, non-chronological backtracking. Since nogoods are added to the original problem, the Dpll algorithm may be restarted once in a while.

2.3.2 Conflict Learning and Non-Chronological Backtracking

Most conflict learning strategies are based on an *implication graph* which is built during the progress of propagation and search. It records the responsible clause for each assignment of a literal performed by the unit propagation or the literals which are fixed by the search. Once an unsatisfiable clause, *i.e.*, all literals in this clause are *false*, is encountered, the conflict learning starts from this clause. It applies the resolution rule on the propagated clauses in reverse and terminates when a pre-defined condition holds.

In more detail, the implication graph is a digraph in which nodes represent fixed *true* literals, *i.e.*, literals assigned *true*. For brevity, no distinction is made between

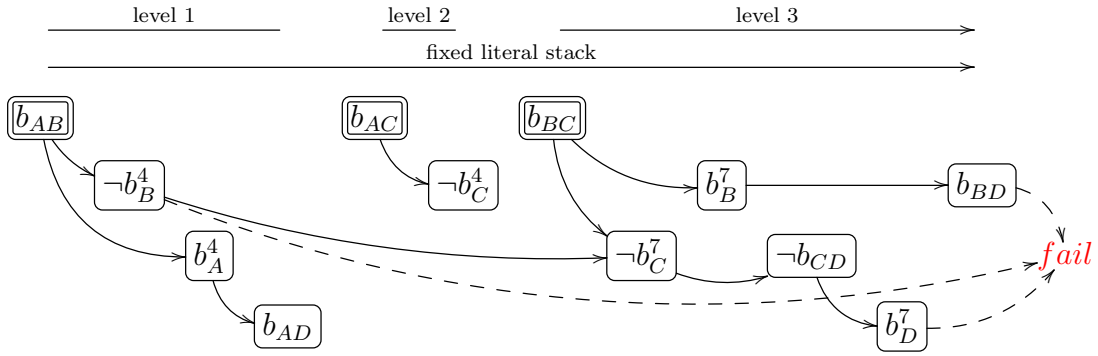


Figure 2.2: The implication graph for Ex. 2.18.

literals and nodes in the remainder of this chapter. All incoming edges to a literal represent the clause responsible for the unit propagation of the literal. Moreover, the *true* literals are kept in a stack that shows the chronological order of their assignment, and are marked by the decision level in which they were assigned.

Example 2.18. Consider the SAT problem from Ex. 2.16 and the search from Ex. 2.17. At first, the search assigns *true* to b_{AB} and then to b_{AC}. After b_{BC} is assigned *true* the propagation detects the conflict in the clause $\neg b_{BD} \vee [\![s_B \leq 4]\!] \vee \neg[\![s_D \leq 7]\!]$.

Figure 2.2 shows a part of the implication graph that was built until the conflict was detected. The doubly boxed literals, e.g. b_{AB}, were fixed by the search and the the single boxed literals, e.g. b_{AD}, by the unit propagation. Solid (incoming) edges show the reason why the literal was fixed to *true*, e.g. the literal $\neg b_C^7$ was fixed because of the *true* value of $\neg b_B^4$ and b_{BC} due to the corresponding clause $\neg b_{BC} \vee b_B^4 \vee b_C^7$. The dotted lines to the *fail* node indicate the reason of the unsatisfiability of the conflicting clause. Moreover, the line above the implication graph indicates the direction of the fixed literal stack from the left to the right. □

Conflict learning starts with the conflicting clause that defines a first tentative nogood. Then, in general, literals in the tentative nogood are replaced one by one by the (negated) literals from the incoming edges of their (negated) literals, in the reverse order that these literals were added to the implication graph. This process continues until the termination criteria is reached.

While most SAT solvers perform the resolution as described in the previous paragraph, they differ on the termination criteria. The first unique implication point (1-UIP) criterion (Zhang *et al.* 2001) which is used in this work is one of the most popular ones. The resolution stops when the tentative nogood only contains one lit-

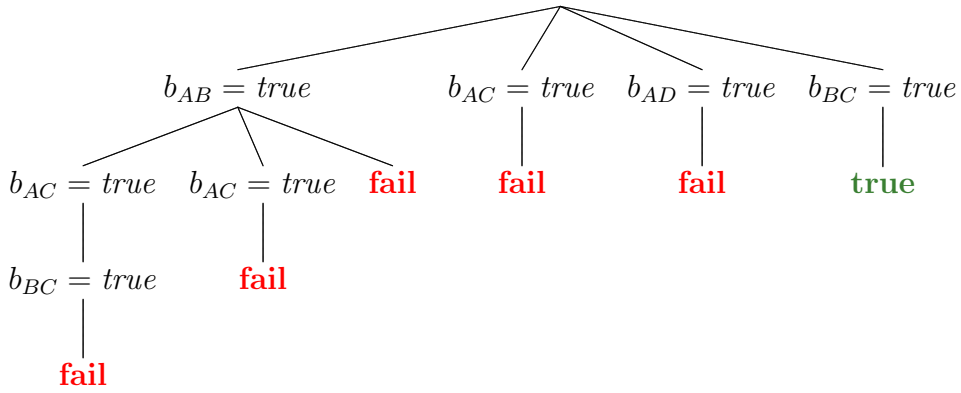


Figure 2.3: The corresponding search tree for Ex. 2.20.

eral from the current decision level. This tentative nogood becomes the final nogood which is (permanently or temporarily) asserted in the set of constraints.

Example 2.19. Consider the running example and the implication graph (Fig. 2.2) that was built until the conflicting clause $\neg b_{BD} \vee b_B^4 \vee \neg b_D^7$ was detected. This is the first tentative nogood. Literal $\neg b_{BD}$ was assigned last. Hence, $\neg b_{BD}$ is replaced by the negated predecessor node $\neg b_B^7$ from the node b_{BD}, or in other words, a resolution step is performed on the tentative nogood and the clause $\neg b_{BD} \vee \neg b_B^7$ which was responsible for the *true* assignment to b_{BD}. The new tentative nogood is $\neg b_B^7 \vee b_B^4 \vee \neg b_D^7$. Then $\neg b_D^7$ is substituted with b_{CD} which leads to the next tentative nogood $\neg b_B^7 \vee b_B^4 \vee \neg b_{CD}$. Next b_{CD} is replaced by b_C^7, $\neg b_B^7$ by b_{BC}, and b_C^7 by s_B^4 which leads to the tentative nogood $s_B^4 \vee \neg b_{BC}$. This tentative nogood is the 1-UIP, since b_{BC} is associated with the current decision level, but not s_B^4. □

Once a nogood is generated and added to the set of constraints, it is used for non-chronological backtracking. In the case of 1-UIP nogoods, it works as follows: the nogood contains exactly one literal from the current decision level and it is not satisfied in this level. The search backjumps to that level (called the *assertion* level) in which the nogood is unit, *i.e.*, the level of the literal from the second highest decision level in the nogood.

In the assertion level, the propagation is reinitiated, but this time the set of constraints includes the added nogood. Since the nogood is unit, the literal from the previous conflict level is assigned to the opposite value. In this way, the search tree is pruned and avoids a re-exploration of inconsistent subtrees.

Example 2.20. Consider the running example. The conflict learning deduced the nogood $s_B^4 \vee \neg b_{BC}$. The conflict level of the nogood is 3 (because of $\neg b_{BC}$) and the assertion level 1 (because of s_B^4). Thus, the search backjumps to the decision

level 1 and undoes all propagation and decisions made in the decision levels 2 and 3. Reaching the decision level, it reinitiates the propagation including the nogood. The propagation and the search steps after the backjump are as follows:

1. Backjumping to decision level 1 and resuming propagation.
   - → Unit propagation on the nogood $b_B^4 \vee \neg b_{BC}$: $b_{BC} = \mathit{false}$.
   - → Unit propagation on $b_{BC} \vee b_C^7$: $b_C^7 = \mathit{true}$, *i.e.*, $s_C \leq 7$.
   - → Unit propagation on $b_{CD} \vee \neg b_C^7$: $b_{CD} = \mathit{true}$.
2. b_2 is assigned *true*.
   - → Unit propagation leads to the conflict clause: $b_{BD} \vee \neg b_D^6$.
   - → Conflict learning (1-UIP nogood): $\neg b_{CD} \vee b_{BC} \vee b_C^4$
   - → Backjumping to the decision level 1.
1. Resuming propagation.
   - → Unit propagation leads to the conflict clause: $b_{BD} \vee \neg b_D^7$.
   - → Conflict learning (1-UIP nogood): $\neg b_{AB}$
   - → Backjumping to the root level and resuming propagation.

Figure 2.3 depicts the search tree when the SAT solver uses 1-UIP conflict learning and backjumping. After five failures and seven explored nodes the SAT solver proves the satisfiability of the instance. Thus, it needs four explored nodes less than the FD solver (cf. Ex. 2.9). □

2.3.3 Conflict-driven Search and Restarts

Conflict driven searches are generic heuristics. Their idea is to exploit the information retrieved from a conflict in order to guide search and avoid exploration of infeasible parts of the search space. Moreover, their aim is to perform well on average on any input. In this work, an activity-based search is used as a conflict-driven search. It is a variant of the variable state independent, decaying sum (VSIDS). Benchmark results by Moskewicz *et al.* (2001) show that VSIDS performs better on average on hard problems than other heuristics.

VSIDS or variants of it are used in many modern SAT solvers implementing the DPLL algorithm. It is based on dynamic activity counters for each variable and branches on the variable with the highest activity counter value. These counters are increased when a variable is involved in a conflict, *i.e.*, occurs in the nogood or is eliminated during conflict learning. Periodically, all counters are reduced (thus *decaying*) not only to avoid overflow, but also to give variables involved in recent

conflicts more weight. Consequently, VSIDS branches over variables involved in lots of recent conflicts. In this work, the VSIDS considered initialises all variable counters with the same value and breaks ties during the variable selection by their input order.

Example 2.21. Consider the running example. After the first failure was detected the first tentative nogood was $\neg b_{BD} \vee b_B^4 \vee \neg b_D^7$. Until the 1-UIP nogood was learned $b_B^4 \vee \neg b_{BC}$, the literals b_{CD}, $\neg b_B^7$, b_C^7 were involved in other tentative nogoods. Therefore, the activity counters for the variables b_{BC}, b_{BD}, b_{CD}, b_B^4, b_B^7, b_C^7 and b_D^7 will be increased. □

As written earlier, VSIDS performs better on average on hard problems than other heuristics. However, its performance can significantly differ for a single problem when the input order of variables or constraints changes. For instance, a change in the input order of variables can yield a different branching at the beginning, or a different input order of constraints can change the order of propagation leading to different nogoods. Not only because of that, but also for robustness issues, VSIDS is usually combined with restarts and occasional random branching.

A restart abandons the current search, but all activity counters are kept and some or all nogoods as well. In general, this leads to a different branching in which variables that were often involved in recent conflicts are more likely to be selected earlier in the search. Therefore, a restart also acts as an escape strategy from hard-to-prove infeasible parts of the search tree.

2.4 Lazy Clause Generation

Lazy clause generation (LCG) is a recently developed technology that is a hybrid of FD solving and SAT solving. It was introduced by Ohrimenko *et al.* (2007) (see, also Ohrimenko and Stuckey 2008, Ohrimenko *et al.* 2009) and reengineered by Feydy and Stuckey (2009) (see also Feydy 2010). The key idea in LCG is to run an FD solver, but to build an explanation of the propagations made by the solver by recording them as clauses on a Boolean variable representation of the problem. Hence, as the FD search progresses we lazily create a clausal representation of the problem. The hybrid has the advantages of FD solving, but inherits the SAT solver's ability to create nogoods to drastically reduce search and use a conflict-driven search.

In this work, the reengineered version (Feydy and Stuckey 2009) of LCG solver is used. Its framework is depicted in Fig. 2.4. The figure shows that the FD solver acts as the master solver and the SAT solver as slave in the LCG solver. The model is passed to the FD solver which generates the variable representations, creates the

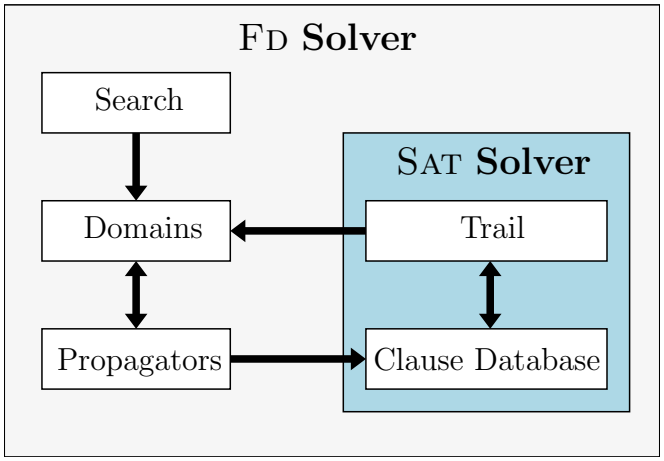


Figure 2.4: Framework of the reengineered version of the LCG generator (adapted from Feydy 2010).

FD propagators for FD constraints, and passes SAT constraints to the internal SAT solver. The LCG search and propagation is controlled by the FD solver in which conflict-driven searches from the SAT solver can be used. The SAT solver acts as an independent propagator with the highest priority. Note that the current version of the LCG solver only supports backtracking, but not backjumping (Feydy 2010).

The communication between the FD and SAT solver works as follows: If an FD propagator updates a variable's domain or infers an empty domain then it explains the reason as clauses. Then, these clauses are passed to the SAT solver which adds them to its clause database and performs unit propagation on them.

2.4.1 Variable Representation

An LCG problem is stated as an FD problem $(\mathcal{C}, \mathcal{X}, \mathcal{D})$, but each variable has a clausal representation in the SAT solver. In the remainder of this work, we use $[\![.]\!]$ as the *names* of Boolean variables. An integer variable $x \in \mathcal{X}$ with the initial domain $D_{init}(x) = [\, l \,..\, u \,]$ is represented by $2(u-l)+1$ Boolean variables $[\![x = l]\!]$, $[\![x = l+1]\!]$, $\ldots$, $[\![x = u]\!]$ and $[\![x \leq l]\!]$, $[\![x \leq l+1]\!]$, $\ldots$, $[\![x \leq u-1]\!]$ where the former is lazily generated (Feydy 2010). The variable $[\![x = d]\!]$ is *true* if x takes the value d, and *false* for a different value of d. Similarly, the variable $[\![x \leq d]\!]$ is *true* if x takes a value less than or equal to d, and *false* for a value greater than d. Sometimes notations $[\![d \leq x]\!]$, $[\![d < x]\!]$, $[\![x < d]\!]$ are used for the literals $\neg[\![x \leq d-1]\!]$, $\neg[\![x \leq d]\!]$, $[\![x \leq d-1]\!]$, respectively.

Not every assignment of Boolean variables is consistent with the integer variable x, for example $\{[\![x = 3]\!], [\![x \leq 2]\!]\}$, *i.e.*, both Boolean variables are true, requires that x is both 3 and ≤ 2. In order to ensure that assignments represent a consistent set of

possibilities for the integer variable x we add to the SAT solver the clauses $DOM(x)$ that encode

$$\begin{aligned}
&[\![x = d]\!] \leftrightarrow [\![x \leq d]\!] && d = l,\\
&[\![x = d]\!] \leftrightarrow \neg[\![x \leq d-1]\!] \wedge [\![x \leq d]\!] && l < d < u,\\
&[\![x = d]\!] \leftrightarrow \neg[\![x \leq d-1]\!] && d = u,\\
&[\![x \leq d]\!] \rightarrow [\![x \leq d+1]\!] && l \leq d < u-1,
\end{aligned}$$

where $D_{init}(x) = [\,l\,..\,u\,]$. Thus, $4(u-l)+1$ clauses are created or just $u-l-1$ if just $[\![x \leq d]\!]$ are used. We let $DOM = \cup\{DOM(x) \mid x \in \mathcal{X}\}$.

Any assignment A on these Boolean variables can be converted to a domain:

$$domain(A)(x) = \{d \in D_{init}(x) \mid \forall [\![c]\!] \in A, vars([\![c]\!]) = \{x\} : x = d \models c\},$$

that is, the domain includes all values for x that are consistent with all the Boolean variables related to x. It should be noted that the domain may assign no values to some variable.

Example 2.22. Consider the one-machine problem discussed in Ex. 2.9 and assume an initial domain $D_{init} = \{s_A \in [\,0\,..\,9\,], s_B \in [\,1\,..\,9\,], s_C \in [\,1\,..\,10\,], s_D \in [\,6\,..\,9\,]\}$. The assignment $\theta = \{\neg[\![s_A \leq 1]\!], \neg[\![s_A = 3]\!], \neg[\![s_A = 4]\!], [\![s_A \leq 6]\!], \neg[\![s_B \leq 2]\!], [\![s_B \leq 5]\!], \neg[\![s_C \leq 4]\!], [\![s_C \leq 7]\!]\}$ is consistent with $x_1 = 2$, $x_2 = 5$, and $x_1 = 6$. Hence $domain(A)(s_A) = \{2, 5, 6\}$. For the remaining variables $domain(A)(s_B) = [\,3\,..\,5\,]$, $domain(A)(s_C) = [\,5\,..\,7\,]$, and $domain(A)(s_D) = [\,6\,..\,9\,]$. Note that for brevity θ is not a fixpoint of unit propagation for $DOM(s_A)$ since we are missing many implied literals such as $\neg[\![s_A = 0]\!]$, $\neg[\![s_A = 8]\!]$ etc. □

2.4.2 Explaining Propagators

In LCG a propagator is extended from a mapping from domains to domains to a generator of clauses describing propagation. When $f(D) \neq D$ we assume the propagator f can determine a clause C to explain each domain change. Similarly, when $f(D)$ is a false domain the propagator must create a clause C that explains the failure.

Example 2.23. Consider the propagator f for the precedence constraint $b_{AB} \leftrightarrow s_A + 5 \leq s_B$ from Ex. 2.9. When applied to the domains $D(b_{AB}) = true$, $D(s_A) = [\,0\,..\,9\,]$ and $D(s_B) = [\,1\,..\,9\,]$ it obtains $f(D)(s_A) = [\,0\,..\,4\,]$, and $f(D)(s_B) = [\,5\,..\,9\,]$. The clausal explanation of the change in domain of s_A is $b_{AB} \wedge [\![s_B \leq 9]\!] \rightarrow [\![s_A \leq 4]\!]$,

similarly the change in domain of s_B is $b_{AB} \wedge \neg[\![s_A \leq -1]\!] \rightarrow \neg[\![s_B \leq 4]\!]$ *i.e.*, $b_{AB} \wedge [\![s_A \geq 0]\!] \rightarrow [\![s_B \geq 5]\!]$. These become the clauses $\neg b_{AB} \vee \neg[\![s_B \leq 9]\!] \vee [\![s_A \leq 4]\!]$ and $\neg b_{AB} \vee [\![s_A \leq -1]\!] \vee \neg[\![s_B \leq 4]\!]$. □

The explaining clauses of the propagation are passed to the SAT solver on which unit propagation is performed. Because the clauses will always have the form $C \rightarrow l$ where C is a conjunction of literals true in the current assignment, and l is a literal not true in the current assignment, the newly added clause will always cause unit propagation, adding l to the current assignment.

Example 2.24. Consider the propagation from Example 2.23. The clauses $\neg b_{AB} \vee \neg[\![s_B \leq 9]\!] \vee [\![s_A \leq 4]\!]$ and $\neg b_{AB} \vee [\![s_A \leq -1]\!] \vee \neg[\![s_B \leq 4]\!]$ are added to the SAT theory. Unit propagation infers that $[\![s_A \leq 4]\!] = \mathit{true}$ and $\neg[\![s_B \leq 4]\!] = \mathit{true}$ since $\neg b_{AB}$, $\neg[\![s_B \leq 9]\!]$ and $[\![s_a \leq -1]\!]$ are *false*, and adds these literals to the assignment. Note that the unit propagation is not finished, since for example the implied literal $[\![s_A \leq 8]\!]$, can be detected *true* as well. □

The unit propagation on the added clauses C is guaranteed to be as strong as the propagator f on the original domains, *i.e.*, if $\mathit{domain}(A) \subseteq D$ then $\mathit{domain}(A') \subseteq f(D)$ where A' is the resulting assignment after addition of C and unit propagation (see Ohrimenko *et al.* 2009).

Note that a single new propagation may be explainable using different set of clauses. In order to get maximum benefit from the explanation we desire a "strongest" explanation as much as possible. A set of clauses C_1 is *stronger* than a set of clauses C_2 if C_2 implies C_1. In other words, C_1 restricts the search space at least as much as C_2.

Example 2.25. Consider the propagator f for the precedence constraint $\neg b_{AD} \leftrightarrow s_D + 3 \leq s_A$ from Ex. 2.9. When applied to the domains $D(b_{AD}) = \{\mathit{true}, \mathit{false}\}$, $D(s_A) = [\,0\,..\,4\,]$ and $D(s_D) = [\,6\,..\,9\,]$ it removes *false* from the domain of b_{AD}. A naïve explanation would only consider the current bounds of s_A and s_D leading to the explanation $[\![s_A \leq 4]\!] \wedge [\![6 \leq s_D]\!] \rightarrow b_{AD}$. We can observe that the same conclusion also holds for an upper bound in $[\,5\,..\,8\,]$ of s_A. Therefore, a stronger explanation is obtained by replacing the literal $[\![s_A \leq 4]\!]$ by $[\![s_A \leq 8]\!]$. This results to the stronger explanation $[\![s_A \leq 8]\!] \wedge [\![6 \leq s_D]\!] \rightarrow b_{AD}$. Note that the literal $[\![6 \leq s_D]\!]$ is always *true*, thus it can be omitted. □

Solving an Example

As explained in the introduction an LCG solver can access search algorithms from the FD solver and SAT solver. This makes an LCG solver versatile in tackling different

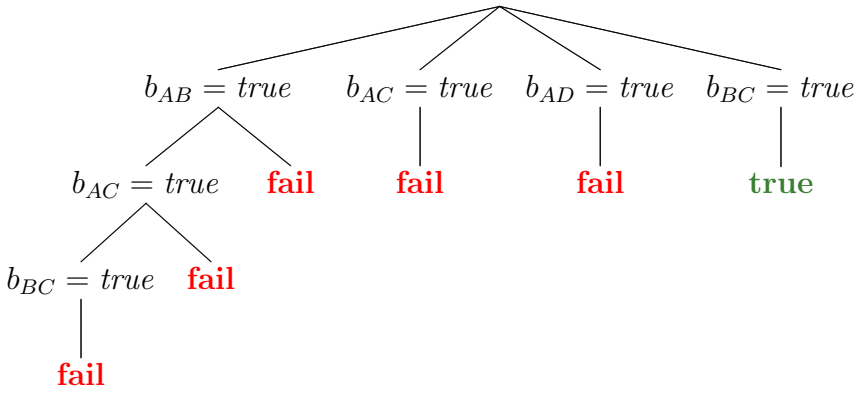


Figure 2.5: The corresponding search tree for Ex. 2.26.

kinds of problem. Here, we show the solving of Ex. 2.9 in order to demonstrate the difference between an FD solver and an LCG solver.

Example 2.26. Consider the one-machine scheduling problem with four tasks A, B, C, and D from Ex. 2.9 on page 15. The goal is to prove the satisfiability of the problem given the variables $b_{AB}, b_{AC}, b_{AD}, b_{BC}, b_{BD}, b_{CD}, s_A, s_B, s_C$, and s_D with their initial domains $\mathcal{D}(b_{ij}) = \{true, false\}$ $(\forall i, j \in \{A, B, C, D\}, i < j)$, $\mathcal{D}(s_A) = [0\,..\,9]$, $\mathcal{D}(s_B) = [1\,..\,9]$, $\mathcal{D}(s_B) = [1\,..\,10]$, $\mathcal{D}(s_B) = [6\,..\,9]$.

The problem is solved by Alg. 2.1 with the same branching procedure: in each node an unfixed variable is selected in the order as stated above and its current domain is branched by enumeration of the elements starting with the *true* element for Boolean variables and the smallest element for integer variables. The first few steps of the algorithm is outlined in the following and the search tree is shown in Fig. 2.5.

1. b_{AB} is assigned *true*.
   - → Propagation on C_1: $f(D)(s_B) = [5\,..\,9]$ with $b_{AB} \wedge [\![0 \leq s_A]\!] \rightarrow [\![5 \leq s_B]\!]$ and $f(D)(s_A) = [0\,..\,4]$ with $b_{AB} \wedge [\![s_B \leq 9]\!] \rightarrow [\![s_A \leq 4]\!]$.
   - → Propagation on C_6: $f(D)(b_{AD}) = \{true\}$ with $[\![s_A \leq 4]\!] \wedge [\![6 \leq s_D]\!] \rightarrow b_{AD}$.
2. b_{AC} is assigned *true*.
   - → Propagation on C_3: $f(D)(s_C) = [5\,..\,10]$ with $b_{AC} \wedge [\![0 \leq s_A]\!] \rightarrow [\![5 \leq s_C]\!]$.
3. b_{BC} is assigned *true*.
   - → Propagation on C_7: $f(D)(s_C) = [8\,..\,10]$ with $b_{BC} \wedge [\![5 \leq s_B]\!] \rightarrow [\![8 \leq s_C]\!]$ and $f(D)(s_B) = [5\,..\,7]$ with $b_{BC} \rightarrow [\![s_B \leq 7]\!]$.
   - → Propagation on C_{11}: $f(D)(b_{CD}) = \{false\}$ with $[\![8 \leq s_C]\!] \rightarrow \neg b_{CD}$.
   - → Propagation on C_{12}: $f(D)(s_D) = [6\,..\,7]$ with $\neg b_{CD} \rightarrow [\![s_D \leq 7]\!]$.

→ Propagation on C_{10}: $f(D)(b_{BD}) = \{true\}$ with $[\![s_B \leq 7]\!] \rightarrow b_{BD}$.
→ Propagation on C_9: $f(D)(s_D) = \emptyset$ with $[\![5 \leq s_B]\!] \wedge b_{BD} \rightarrow [\![8 \leq s_D]\!]$.
→ Conflict clause: $\neg[\![5 \leq s_B]\!] \vee \neg b_{BD} \vee [\![8 \leq s_D]\!]$.
→ Conflict learning (1-UIP nogood): $\neg[\![5 \leq s_B]\!] \vee \neg b_{BC}$.
→ Backtracking.

2. Resuming propagation.

→ Propagation on latest nogood: $f(D)(b_{BC}) = \{false\}$ with $[\![5 \leq s_B]\!] \rightarrow \neg b_{BC}$.
→ Propagation on C_8: $f(D)(s_C) = [\,1\,..\,7\,]$ with $\neg b_{BC} \rightarrow [\![s_C \leq 7]\!]$.
→ Propagation on C_{12}: $f(D)(b_{BC}) = \{true\}$ with $[\![s_C \leq 7]\!] \rightarrow b_{CD}$.
→ Propagation on C_8: $f(D)(s_B) = [\,7\,..\,9\,]$ with $\neg b_{BC} \wedge [\![5 \leq s_C]\!] \rightarrow [\![7 \neq s_B]\!]$.
→ Propagation on C_{11}: $f(D)(s_D) = [\,7\,..\,9\,]$ with $\neg b_{CD} \wedge [\![5 \leq s_C]\!] \rightarrow [\![7 \neq s_D]\!]$.
→ Propagation on C_9: $f(D)(b_{BD}) = \{false\}$ with $[\![7 \leq s_B]\!] \rightarrow \neg b_{BD}$.
→ Propagation on C_{10}: $f(D)(s_D) = \emptyset$ with $\neg b_{BD} \rightarrow s_D \leq 6$.
→ Conflict clause: $b_{BD} \rightarrow s_D \leq 6$.
→ Conflict learning (1-UIP): $\neg[\![5 \leq s_B]\!] \vee \neg[\![5 \leq s_C]\!]$.
→ Backtracking.

1. Resuming propagation.

→ Propagation leads to a conflict.
→ Conflict learning (1-UIP nogood): $\neg b_{AB}$.
→ Backtracking to the root level.

0. Resumption of the initial propagation.
1. b_{AC} is assigned *true*.

→ Propagation leads to a conflict.
→ Conflict learning (1-UIP nogood): $\neg[\![4 \leq s_A]\!] \vee \neg b_{AC}$.
→ Backtracking to the root level and resumption of the initial propagation.

After five failures and six explored nodes the LCG solver proves the satisfiability of the instance. Thus, it needs five explored nodes less than the FD solver (cf. Ex. 2.9). □

2.4.3 Conflict-driven Search and Optimisation

In order to use a conflict-driven search, in our case a variant of VSIDS, in an LCG solver, we ask the SAT solver what its preferred literal is for branching on. This corresponds to an atomic constraint $x \leq d$ or $x = d$ and we branch on $x \leq d \vee x > d$ or $x = d \vee x \neq d$. Note that the search is still controlled by the FD search engine, so that we use its standard approach for the branch-and-bound algorithm to implement the optimisation search.

Normally SAT solvers use dichotomic restart search for optimisation as the SAT solver itself does not have optimisation search built in. This leads to a situation when a satisfaction search fails, but the inferred nogoods are not valid for subsequent searches. The reason for this it that they assume a wrong upper bound on the objective.

The combination of branch-and-bound algorithm and VSIDS is much stronger, since in the continuation of the search with a better bound, the activity counts at the time of finding a new better solution are used in the same part of the search tree, and all inferred nogoods remain valid. Note that if the distance of the lower and upper bound on the objective is large, a dichotomic search may find quicker a solution than a branch-and-bound algorithm.

2.4.4 Background

As indicated earlier in this section, LCG was developed to combine the strengths of FD and SAT solvers. For FD solvers it is their high-level view of problems and their flexible framework whereas for SAT solvers it is their nogood technologies and conflict-driven searches. One important aspect in which LCG differs from SAT solvers or nogood technologies in FD solvers, is the representation of domains with inequalities. These inequalities allow an efficient way for bound propagators to explain their propagation.

The first version of LCG (Ohrimenko *et al.* 2007) was the predecessor of the reengineered version used in this thesis. In this first version, the SAT solver is the master whereas the FD solver the slave. Consequently, no FD search could be used to drive the search. Moreover, the integration of FD propagators was not simple.

The reengineered version (Feydy and Stuckey 2009) turned the master-slave principle around which enabled not only the use of customisable FD searches, but also made the integration of FD propagators easier, as they only had to be extended with explanations. But still this version has limitations.

The main limitation is the eager generation of the inequalities' literals which can

lead to inefficiency in the SAT solver due to the large size of the clausal representation of the domains. A solution would be not only to create those literals on demand, but also to delete them when they are not needed anymore.

Another limitation is the eager explanation of constraint propagation which rapidly increases the size of the clause database. These explanations are used in the conflict analysis and for short circuiting propagation in subsequent search. If we concentrate on the first purpose then delaying the production of the explanation to the time of the conflict analysis avoids the necessity of adding explanations to the clause database and the building of explanations that are not related to the conflict.

Future versions of LCG will tackle these and other weaknesses as described to make LCG an even more efficient solver than it already is.

2.5 Satisfiability Modulo Theories

Satisfiability modulo theories solving (Sebastiani 2007, Nieuwenhuis *et al.* 2006, Kroening and Strichman 2008, Barrett *et al.* 2009) is an extension of SAT solving. Instead of checking the satisfiability of a formula in propositional logic as in SAT solving, a formula in first-order logic is checked with respect to some given background theories T. First-order logic formulas are enriched by quantifiable variables and non-logical symbols as predicates (e.g. $x < y$), functions (e.g. $x + y$), and constants (e.g. 0 and 87).

Definition 2.17 (Satisfiability Modulo Theory Problem)**.** A *satisfiability modulo theory* (SMT) problem is a pair (Φ, T) where Φ is a formula in first-order logic and T a background theory that determines how to interpret atomic predicates appearing in Φ.

An *atomic predicate* is a formula in Φ that contains no logical symbol such as $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ and $\neg$. Note that a function can be transformed to a predicate and a Boolean variable can be seen as a predicate with an arity of 0.

An SMT problem (Φ, T) is satisfiable modulo the theory T if an assignment θ exists for the atomic predicates in Φ that satisfies Φ modulo T.

The following example is a running example throughout this section.

Example 2.27. Consider the one-machine scheduling problem with four tasks A, B, C, and D from Ex. 2.9 on page 15.

This problem can be formulated as an SMT problem (Φ, T) where T is the theory

of difference constraints and Φ is defined as follows:

$$
\begin{array}{llll}
 & C_1 \equiv (s_A + 5 \leq s_B \vee \neg(s_A - 2 \leq s_B)) & \wedge & C_2 \equiv (s_A + 5 \leq s_C \vee \neg(s_A - 1 \leq s_C)) \\
\wedge & C_3 \equiv (s_A + 5 \leq s_D \vee \neg(s_A - 2 \leq s_D)) & \wedge & C_4 \equiv (s_B + 3 \leq s_C \vee \neg(s_B - 1 \leq s_C)) \\
\wedge & C_5 \equiv (s_B + 3 \leq s_D \vee \neg(s_B - 2 \leq s_D)) & \wedge & C_6 \equiv (s_C + 2 \leq s_D \vee \neg(s_C - 2 \leq s_D)) \\
\wedge & C_7 \equiv 0 \leq s_A & \wedge & C_8 \equiv s_A \leq 9 \\
\wedge & C_9 \equiv 1 \leq s_B & \wedge & C_{10} \equiv s_B \leq 9 \\
\wedge & C_{11} \equiv 1 \leq s_C & \wedge & C_{12} \equiv s_C \leq 10 \\
\wedge & C_{13} \equiv 6 \leq s_D & \wedge & C_{14} \equiv s_D \leq 9
\end{array}
$$

Atomic predicates have the form of constraints $x_1 + d \leq x_2$ where x_1, x_2 are variables and d an integer constant, e.g. $0 \leq s_A$ and $s_A + 5 \leq s_B$. These predicates are written in the canonical form in which the variable of a task with the smallest name appears on the left-hand side. Such canonical form can be achieved easily and is important for deduction of implied constraints for some SMT solvers (see Nieuwenhuis and Oliveras 2005).

Note that the constraints $C_1, C_2, \ldots, C_6$ refer to the constraints in a CSP model and the domains of the start times are implicitly modelled by the remaining constraints rather than explicitly as in a CSP model. □

Applications for SMT can be found in software and hardware verification, scheduling, model checking, equivalence checking, test generation, and predicate abstraction amongst others (see, e.g., Bofill *et al.* 2008a, Tillmann and de Halleux 2008, Armando *et al.* 2009, Lahiri *et al.* 2006).

The SMT community has implemented theory solvers for many theories. Some of them are of particular interest such as integer and real arithmetic (e.g. difference logic (Nieuwenhuis and Oliveras 2005, Cotton and Maler 2006), unit-two-variable per inequality (Lahiri and Musuvathi 2005, Seshia *et al.* 2007), linear arithmetic (Dutertre and de Moura 2006), non-linear arithmetic (Fränzle *et al.* 2007, Borralleras *et al.* 2009)) and uninterpreted functions with equality, arrays, and fixed-width bit vectors (see, e.g., Pnueli *et al.* 1999, Meir and Strichman 2005, Ganesh and Dill 2007, Bofill *et al.* 2008b, Brummayer and Biere 2009, Bryant *et al.* 2009). Normally, an SMT problem contains a combination of theories, although such a combination may not be straightforward (see, e.g., Manna and Zarba 2003, Bozzano *et al.* 2005).

2.5.1 Solving

Different kinds of SMT solver exist which are grouped as eager and lazy solvers. The latter solvers are further refined into online and offline solvers. Most modern solvers are lazy online SMT solvers.

An *eager* SMT solver transforms the first-order formula Φ modulo the theory T into an equisatisfiable propositional formula Φ' where *equisatisfiable* means $\Phi \leftrightarrow \Phi'$. Then a SAT solver (see Sec. 2.3) solves the transformed formula Φ'. This has the advantage that the best available SAT solver can be taken and the SAT solver can exploit the full knowledge about the problem from the beginning. However, a sophisticated encoding is needed for each theory which does not break the implicit size bounds of the SAT solver.

Definition 2.18 (Propositional Abstraction)**.** Consider the SMT problem (Φ, T). The *propositional abstraction* Φ_P of Φ is a formula where a new Boolean variable denoted by $[\![p]\!]$ is substituted for each interpreted atomic predicate p in Φ.

Then a formula Φ can be rewritten as $\Phi_P \wedge \Phi_T$ where $\Phi_T = \bigwedge_{p \in Pr} p \leftrightarrow [\![p]\!]$ and Pr is the set of interpreted atomic predicates involved in the theory T. This means that the formula can be equisatisfiably transformed into a formula with propositional logic part Φ_P and first-order logic part Φ_T modulo the background theory T where the atomic predicates in Φ_T appear in a reified context. A solution for $\Phi_P \wedge \Phi_T$ is an assignment of the Boolean variables in Φ_P that satisfied Φ_P as well as Φ_T.

Example 2.28. Consider the running example Ex. 2.27. The propositional abstraction of Φ is

$$\begin{aligned}(b_{AB} \vee \neg b^*_{BA}) \wedge (b_{AC} \vee \neg b^*_{CA}) \wedge (b_{AD} \vee \neg b^*_{DA}) \wedge (b_{BC} \vee \neg b^*_{CB}) \\ \wedge (b_{BD} \vee \neg b^*_{DB}) \wedge (b_{CD} \vee \neg b^*_{DC}) \wedge [\![0 \leq s_A]\!] \wedge [\![s_A \leq 9]\!] \wedge [\![1 \leq s_B]\!] \\ \wedge [\![s_B \leq 9]\!] \wedge [\![1 \leq s_C]\!] \wedge [\![s_C \leq 10]\!] \wedge [\![6 \leq s_D]\!] \wedge [\![s_D \leq 9]\!]\end{aligned}$$

where

$$\begin{aligned}&b_{AB} \equiv [\![s_A + 5 \leq s_B]\!] && b^*_{BA} \equiv [\![s_A - 2 \leq s_B]\!] && b_{AC} \equiv [\![s_A + 5 \leq s_C]\!] \\ &b^*_{CA} \equiv [\![s_A - 1 \leq s_C]\!] && b_{AD} \equiv [\![s_A + 5 \leq s_D]\!] && b^*_{DA} \equiv [\![s_A - 2 \leq s_D]\!] \\ &b_{BC} \equiv [\![s_B + 3 \leq s_C]\!] && b^*_{CB} \equiv [\![s_B - 1 \leq s_C]\!] && b_{BD} \equiv [\![s_B + 3 \leq s_D]\!] \\ &b^*_{DB} \equiv [\![s_B - 2 \leq s_D]\!] && b_{CD} \equiv [\![s_C + 2 \leq s_D]\!] && b^*_{DC} \equiv [\![s_C - 2 \leq s_D]\!]. \quad \square\end{aligned}$$

Instead of fully transforming the SMT problem into an equisatisfiable SAT problem, the *lazy* SMT solver builds a propositional abstraction of the problem and

Algorithm 2.3: *offline_solve*(Φ, T) — scheme of offline SMT solver

Input: SMT $(\Phi_P \wedge \Phi_T, T)$
Output: *true*, if $\Phi_P \wedge \Phi_T$ is satisfiable modulo theory T; otherwise *false*

```
1 repeat
2 |   α := sat_solver(Φ_P);
3 |   if α = unsat then return false;
4 |   M := theory_consistency_check(Φ_T, T, α);
5 |   if M ≠ sat then
6 |   |   add ¬(M) permanently to Φ_P;
7 until M = sat;
8 return true;
```

passes it to the SAT solver. The SAT solver proposes partial or full assignments to the SMT solver which are checked for consistency modulo the theory T by a theory solver, T-solver, for short. Furthermore, the T-solver tries to extend the assignment regarding the theory T. In both cases the T-solver generates explanations for its decision, which are added the SAT solver.

The interaction between the SAT solver and T-solver differs in the offline and online cases. In first case, the SAT solver is used as a black-box solver and in the second case, both are tightly connected.

In general, offline solvers (see Alg. 2.3) work as follows: the SAT solver finds a solution for Φ_P (line 2) and then the theory T-solver validates the assignment if it also is a solution for Φ_T (line 4). If so then the SMT problem is satisfiable; otherwise the T-solver adds (conflict) clauses to the SAT solver (line 6) and restarts the SAT solver. This process is repeatedly applied until a solution is found or the SAT solver proves the unsatisfiability.

Example 2.29. Consider the running Ex. 2.27 with the propositional abstraction in Ex. 2.28. Assume the SAT solver assigns *true* or *false* to the Boolean variables in this order: b_{AB}, b^*_{BA}, b_{AC}, b^*_{CA}, b_{AD}, b^*_{DA}, b_{BC}, b^*_{CB}, b_{BD}, b_{DB^*}, b_{CD}, and b^*_{DC}. All remaining Boolean variables appear in unit clauses, for this reason the SAT solver always assigns *true* to them. The first three steps of Alg. 2.3 are shown below in which the assignment of the "always" true Boolean variables are omitted:

1. SAT solver proposes $\{b_{AB} \mapsto \mathit{true}, b^*_{BA} \mapsto \mathit{true}, b_{AC} \mapsto \mathit{true}, b^*_{CA} \mapsto \mathit{true}, b_{AD} \mapsto \mathit{true}, b^*_{DA} \mapsto \mathit{true}, b_{BC} \mapsto \mathit{true}, b^*_{CB} \mapsto \mathit{true}, b_{BD} \mapsto \mathit{true}, b^*_{DB} \mapsto \mathit{true}, b_{CD} \mapsto \mathit{true}, b^*_{DC} \mapsto \mathit{true}\}$.

   $\rightarrow$ T-solver detects conflict and returns $\neg[\![0 \leq s_A]\!] \vee \neg b_{AB} \vee \neg b_{BC} \vee \neg b_{CD} \vee [\![s_D \leq 9]\!]$.

→ Clause $\neg[\![0 \leq s_A]\!] \vee \neg b_{AB} \vee \neg b_{BC} \vee \neg b_{CD} \vee [\![s_D \leq 9]\!]$ is added to Φ_P.

2. SAT solver proposes an assignment that differs from the previous one in: $b_{CD} \mapsto \mathit{false}$.

   → T-solver detects conflict and returns $b_{CD} \vee \neg b^*_{BC}$.

   → Clause $b_{CD} \vee \neg b^*_{DC}$ is added to Φ_P.

3. SAT solver proposes an assignment that differs from the previous one in: $b^*_{DC} \mapsto \mathit{false}$.

   → T-solver detects conflict and returns $\neg[\![0 \leq s_A]\!] \vee \neg b_{AB} \vee \neg b_{BD} \vee b^*_{DC} \vee [\![s_C \leq 10]\!]$.

   → Clause $\neg[\![0 \leq s_A]\!] \vee \neg b_{AB} \vee \neg b_{BC} \vee \neg b_{CD} \vee [\![s_D \leq 9]\!]$ is added to Φ_P.

In the second step the SMT solver learns about the relationship between b_{CD} and b^*_{DC} whereas in the first and third step, it learns about an invalid ordering of the tasks regarding their earliest start times and latest completion time. □

The advantage of offline solvers are their modularity and flexibility, but the SAT search is not driven by information from the theory.

The interaction between the SAT solver and T-solver is more fine grained in online solvers: instead of waiting for a solution for Φ_P from the SAT solver, the T-solver checks if partial assignments are consistent modulo theory T. Additional, theory propagators are used in order to infer Boolean variables implied by the current (partial) assignment.

Nowadays, state-of-the-art SMT solvers are online lazy SMT solvers that implement an extension of the SAT DPLL algorithm (see Alg. 2.2) referred to as the DPLL(T) algorithm (Nieuwenhuis *et al.* 2005, 2006). A basic scheme with chronological backtracking is given in Alg. 2.4 and works as follows: at first, the algorithm enters a propagation loop (lines 1 to 11) which consists of SAT unit propagation on Φ_P (line 2), theory consistency check (line 4), and theory propagation (line 4). The unit propagation returns either a partial assignment or *unsat* if a conflicting clause was detected. In the latter case, the algorithm returns *false* and backtracks to the previous decision level. In the former case, the T-solver checks if the partial assignment is still consistent modulo theory T. If the assignment is inconsistent then the consistency check returns a minimal conflict set M from the partial assignment which is added to the clause database of the SAT solver. Then the backtracking is initiated (lines 5 to 7). If the assignment is consistent and a solution of the propositional formula Φ_P then the original SMT problem is satisfiable modulo T (line 8). If it is not a solution then the T-solver tries to infer implied literals $[\![p]\!]$ where p is

Algorithm 2.4: *dpll_solve*(Φ, T, α) — scheme of DPLL(T) algorithm with chronological backtracking

Input: SMT $(\Phi_P \wedge \Phi_T, T)$
Output: *true*, if $\Phi_P \wedge \Phi_T$ is satisfiable modulo theory T; otherwise *false*

```
1  repeat
2      α := unit_propagate(Φ_P, α);
3      if α = unsat then return false;
4      M := theory_consistency_check(Φ_T, T, α);
5      if M ≠ sat then
6          add ¬(M) permanently to Φ_P;
7          return false;
8      if ∀b ∈ B' : |D'(b)| = 1 then return true;
9      (M, α) := theory_propagate(Φ_T, T, α);
10     Φ'_P := Φ_P ∧ M;
11 until M = ∅;
12 l := propose_literal(Φ'_P);
13 return dpll_solve(Φ, T, α + l) or dpll_solve(Φ, T, α + ¬l);
```

an interpreted predicate in Φ_T (line 9). These implied literals extend the partial assignment and a clausal explanation is added to the SAT database. The propagation stops if no implied literal is inferred. Then the SAT solver selects an unfixed literal (line 12) and branches over it (line 13).

In general, an SMT DPLL(T) solver uses conflict learning, conflict-driven search, and non-chronological backtracking.

Example 2.30. Consider the running example Ex. 2.27 with the propositional abstraction in Ex. 2.28 and the SAT assignment order in Ex. 2.29. The DPLL(T) algorithm with backjumping and conflict learning solves the problem as follows:

0. Root level.
   → Unit propagation assigns *true* to $[\![0 \leq s_A]\!]$, $[\![s_A \leq 9]\!]$, $[\![1 \leq s_B]\!]$, $[\![s_B \leq 9]\!]$, $[\![1 \leq s_C]\!]$, $[\![s_C \leq 10]\!]$, $[\![6 \leq s_D]\!]$, and $[\![s_D \leq 9]\!]$.
1. b_{AB} is assigned *true*.
   → T-propagate: $b^*_{BA} = \mathit{true}$ with $b_{AB} \rightarrow b^*_{BA}$.
   → T-propagate: $b^*_{DA} = \mathit{true}$ with $b_{AB} \wedge [\![s_B \leq 9]\!] \wedge [\![6 \leq s_D]\!] \rightarrow b^*_{DA}$.
   → Unit propagation on $b_{AD} \vee \neg b^*_{DA}$: $b_{AD} = \mathit{true}$.
2. b_{AC} is assigned *true*.
   → T-propagate: $b^*_{CA} = \mathit{true}$ with $b_{AC} \rightarrow b^*_{CA}$.
3. b_{BC} is assigned *true*.

- → T-propagate: $b^*_{CB} = \mathit{true}$ with $b_{BC} \rightarrow b^*_{CB}$.
- → T-propagate: $b^*_{DB} = \mathit{true}$ with $b_{BC} \wedge [\![s_C \leq 10]\!] \wedge [\![6 \leq s_D]\!] \rightarrow b^*_{DB}$.
- → Unit propagation on $b_{BD} \vee \neg b^*_{DB}$: $b_{BD} = \mathit{true}$.
- → T-propagate: $b_{CD} = \mathit{false}$ with $[\![s_D \leq 9]\!] \wedge [\![0 \leq s_A]\!] \wedge b_{AB} \wedge b_{BC} \rightarrow \neg b_{CD}$.
- → Unit propagation on $b_{CD} \vee \neg b^*_{CD}$: $b^*_{CD} = \mathit{false}$.
- → T-consistency check (conflict clause): $\neg[\![0 \leq s_A]\!] \vee \neg b_{AB} \vee \neg b_{BD} \vee b^*_{DC} \vee \neg[\![s_C \leq 10]\!]$.
- → Conflict learning (1-UIP nogood): $\neg b_{BC} \vee \neg b_{AB} \vee \neg[\![0 \leq s_A]\!] \vee \neg[\![s_C \leq 10]\!] \vee \neg[\![6 \leq s_D]\!] \vee \neg[\![s_D \leq 9]\!]$.
- → Backjumping to the decision level 1.

1. Resuming propagation.
   - → Unit propagation on latest nogood: $b_{BC} = \mathit{false}$.
   - → Unit propagation on $b_{BC} \vee \neg b^*_{CB}$: $b^*_{CB} = \mathit{false}$.
   - → T-propagate: $b^*_{DC} = \mathit{true}$ with $\neg b^*_{CB} \wedge [\![s_B \leq 9]\!] \wedge [\![6 \leq s_D]\!]$.
   - → Unit propagation on $b_{CD} \vee \neg b^*_{DC}$: $b_{CD} = \mathit{false}$.
2. b_{AC} is assigned *true*.
   - → Unit propagation on $\neg b_{AC} \rightarrow b^*_{CA}$: $b^*_{CA} = \mathit{true}$.
   - → T-propagate: $b_{BD} = \mathit{false}$ with $[\![s_D \leq 9]\!] \wedge [\![0 \leq s_A]\!] \wedge b_{AC} \wedge \neg b^*_{CB} \rightarrow \neg b_{BD}$.
   - → Unit propagation on $b_{BD} \vee \neg b^*_{DB}$: $b^*_{DB} = \mathit{false}$.
   - → T-consistency check (conflict clause): $\neg[\![0 \leq s_A]\!] \vee \neg b_{AC} \vee \neg b_{CD} \vee b^*_{DB} \vee \neg[\![s_B \leq 9]\!]$.
   - → Conflict learning (1-UIP nogood): $\neg b_{AC} \vee \neg b_{CD} \vee b^*_{CB} \vee \neg[\![0 \leq A]\!] \vee \neg[\![s_B \leq 9]\!] \vee [\![s_D \leq 9]\!]$.
   - → Backjumping to the decision level 1.

Here, the SMT solver performs the same search steps as the SAT solver in Ex. 2.20 on page 28. If the LCG solver uses backjumping then the SMT solver also performs the same steps. □

2.5.2 Comparison to Lazy Clause Generation Solving

SMT solving using the DPLL(T) algorithm and LCG solving have strong similarities: (*i*) a SAT solver is used to inherit its advanced techniques for conflict learning, non-chronological backtracking,[1] and conflict-driven search, (*ii*) both lazily translate the original problem to the SAT solver as the search progresses, and (*iii*) the translation is performed by specialised propagators which explain their propagation.

[1] If the LCG solver supports non-chronological backtracking.

Hence, theory solvers can be implemented as LCG propagators and vice versa.

A difference between both solving approaches is that the SAT solver is the master solver in an SMT DPLL(T) solver whereas it is just a propagator in an LCG solver. The consequence is that the SMT solver uses search strategies provided by the SAT solver whereas the LCG solver can additionally use the search strategies from the FD solver.

Another difference is that each variable in the LCG solver has a representation in the SAT solver whereas normally, the SMT DPLL(T) does not, since it is not needed. Instead, each constraint has a representation. Put simply, the SMT solver solves the relational dependencies between constraints whereas the LCG solver tries to construct a solution for each constraint. Consequently, a SAT conflict-driven search can take the variables' domains into account in the LCG solver.

3

Satisfaction and Implication Algorithms for Unit Two Variable Per Inequalities

The unit two variable per inequality (UTVPI) constraints form one of the largest class of integer constraints which are polynomial time solvable (unless P=NP). There is considerable interest in their use for constraint solving (Jaffar *et al.* 1994, Harvey and Stuckey 1997), abstract interpretation (Miné 2006), spatial databases (Sitzmann and Stuckey 2000) and theorem proving (Lahiri and Musuvathi 2005). Moreover, they generalise difference constraints which model precedence constraints in scheduling problems.

Most uses of UTVPI constraint are inherently incremental. UTVPI constraint solving repeatedly asks satisfiability questions in an incremental manner in order to drive a search in a large search space. Abstract interpretation uses of UTVPI (Miné 2006) build descriptions of program points in an incremental manner by taking the description of the previous program point and adding new constraints to generate a description for the next program point. Theorem proving may be non-incremental as in Lahiri and Musuvathi (2005) where UTVPI problems arise as subproblems required by a verification system, but modern techniques such as SMT solving (see Sec. 2.5 on page 37), require incremental satisfaction and implication algorithms as well as algorithms for explanation.

In this chapter we develop new incremental algorithms for UTVPI constraint satisfaction and implication. These algorithms can be directly used in a lazy online SMT solver using the DPLL(T) algorithm (see Alg. 2.4 on page 42) as the theory consistency check and the theory propagator. Moreover, they can be integrated in

an FD solver (without explanation) and an LCG solver.

3.1 Introduction

A UTVPI constraint has the form $ax + by \leq d$ where x, y are integer variables, $d \in \mathbb{Z}$ and $a, b \in \{-1, 0, 1\}$. For example $x + y \leq 2$, $x - y \leq -1$, $0 \leq -1$ and $x \leq 2$ are UTVPI constraints. UTVPI constraint solving is based on transitive closure: The constraints $ax - y \leq d_1$ and $y + bz \leq d_2$ imply the constraint $ax + bz \leq d_1 + d_2$. We can determine all the UTVPI consequences of a set of UTVPI constraints by transitive closure, but we need to *tighten* some constraints. The transitive closure procedure can generate constraints of the form $x + x \leq d$ and $-x - x \leq d$, which need to be tightened to $x \leq \lfloor \frac{d}{2} \rfloor$ and $-x \leq \lfloor \frac{d}{2} \rfloor$ respectively.

Jaffar *et al.* (1994) and Harvey and Stuckey (1997) present incremental consistency checking algorithms for adding a UTVPI constraint c to a set ϕ of UTVPI constraints. They are based on maintaining the transitive and tight closure of the set of UTVPI constraints ϕ involving n variables. Both algorithms require $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space for an incremental satisfaction check. Both algorithms can also be used to incrementally check implication of UTVPI constraints by $\phi \cup \{c\}$. These algorithms require $\mathcal{O}(n^2 + p)$ time and $\mathcal{O}(n^2 + p)$ space for incremental implication checking, where p is the number of constraints we need to check for implication. In order to (non-incrementally) check satisfiability of m UTVPI constraints on n variables these approaches require $\mathcal{O}(n^2 m)$ time, and to check implication they require $\mathcal{O}(n^2 m + p)$ time.

An improvement on the complexity of non-incremental satisfiability for UTVPI constraints was devised by Lahiri and Musuvathi (2005). They define a non-incremental satisfiability algorithm requiring $\mathcal{O}(nm)$ time and $\mathcal{O}(n + m)$ space. The key to their approach is to map UTVPI constraints to difference constraints (also called separation theory constraints) of the form $x - y \leq d$, where x and y are integer variables and $d \in \mathbb{Z}$.

The difference constraints are a well studied class of constraints because of their connection to shortest path problems. We can consider the constraint $x - y \leq d$ as a directed edge $x \rightarrow y$ with weight d. Satisfiability of difference constraints corresponds to the problem of negative weight cycle detection, and implication of difference constraints corresponds to finding shortest paths (see e.g. Cotton and Maler 2006, for details).

The mapping of UTVPI to difference constraints by Lahiri and Musuvathi (2005) is a relaxation of the problem. The relaxed problem is solved by a negative (weight)

cycle detection algorithm but it guarantees only the satisfiability in $\mathbb{Q}$ for the UTVPI problem. In order to check satisfiability in $\mathbb{Z}$ they need to construct an auxiliary graph and check for certain paths in this graph.

In this chapter we first extend Lahiri and Musuvathi's algorithm (see Lahiri and Musuvathi 2005) to check satisfaction incrementally in $\mathcal{O}(n \log n + m)$ and $\mathcal{O}(n+m)$ space. Then we show how to build an incremental satisfiability and implication algorithm using the relaxation of Lahiri and Musuvathi (2005) and the incremental implication approaches for difference constraints of Cotton and Maler (2006), which can incrementally check implication in $\mathcal{O}(n \log n + m + p)$ time and $\mathcal{O}(n + m + p)$ space.

We give experimental results showing that our algorithms improve upon the previous incremental algorithms for UTVPI satisfaction and implication checking, unless the constraint graph is dense.

We then consider using our incremental approach as a basis for a non-incremental algorithm for implication checking in $\mathcal{O}(n^2 \log n + nm + p)$ time and $\mathcal{O}(n + m + p)$ space. Similarly we can generate all implied constraints in $\mathcal{O}(n^2 \log n + nm)$ time and $\mathcal{O}(n+m+p)$ space, where here p is the number of generated implied constraints.

One of the interests of solving UTVPI constraints is in solving Boolean combinations of UTVPI constraints, e.g. $(x - y \leq 3 \vee y - x \leq 4) \rightarrow (x - z \leq 2 \wedge z - y \leq 1)$. Seshia *et al.* (2007) discuss how to encode Boolean combinations of UTVPI constraints in conjunctive normal form (CNF) by giving a tight bound on the region of satisfiability. An alternate approach is to use incremental satisfaction and implication algorithms in a lazy online SMT solver (Nieuwenhuis *et al.* 2005). This requires that one can efficiently discover unsatisfiable subsets and implicants of implied constraints. We discuss how to extend the incremental algorithm to discover minimal unsatisfiable subsets and minimal implicants in $\mathcal{O}(n)$ time. Surprisingly this is not as simple as the case for difference constraints.

In summary the contributions of this chapter are:

- A new incremental satisfiability algorithm for UTVPI based on the approach of Lahiri and Musuvathi, which is asymptotically better than previous algorithms for this problem.

- A new incremental implication algorithm for UTVPI, based on a fundamental new understanding of how we can compute the tightened transitive closure, which is asymptotically better than previous algorithms for this problem.

- Experiments illustrating that for sparse problems our algorithms are significantly better than existing algorithms for these problems.

- New non-incremental algorithms for implication checking and implication generation which are asymptotically better than existing algorithms for these problems.
- The first algorithms we are aware of specifically for generation of minimal unsatisfiable subsets and minimal implicants for UTVPI problems.

The remainder of the chapter is organised as follows. In the next section we give preliminary definitions. In Sec. 3.3 we explain the approach of Lahiri and Musuvathi (2005) to UTVPI satisfaction. In Sec. 3.4 we show how their approach can be modified to perform incremental satisfaction. In Sec. 3.5 we show how to do incremental implication checking, which also introduces a new way to do incremental satisfiability checking. In Sec. 3.6 we give experimental comparisons of the algorithms for satisfiability and implication. In Sec. 3.7 we show how we can create non-incremental implication checking from the incremental algorithms. In Sec. 3.8 we explain how to generate minimal explanations of the unsatisfiability and implication for use in an SMT solver. Finally in Sec. 3.9 we conclude.

3.2 Preliminaries

In this section we give notations and preliminary concepts.

A *weighted directed graph* $G = (V, E)$ is made up of vertices V and a set E of weighted directed edges (u, v, d) from vertex $u \in V$ to vertex $v \in V$ with weight d. We also use the notation $u \xrightarrow{d} v$ to denote the edge (u, v, d).

A *path* P from v_0 to v_k in graph G, denoted $v_0 \rightsquigarrow v_k$, is a sequence of edges $e_1, \ldots, e_k$ where $e_i = (v_{i-1}, v_i, d_i) \in E$ and $k \in \mathbb{N}$. Let $|P|$ be the number of edges appearing in P. A *simple path* P is a path where $v_i \neq v_j, 0 \leq i < j \leq k$.

A (simple) *cycle* P is a path P where $v_0 = v_k$ and $v_i \neq v_j, 0 \leq i < j \wedge k \wedge (i \neq 0 \vee j \neq k)$.

The *path weight* of a path P, denoted $w(P)$ is $\Sigma_{i=1}^{k} d_i$.

Let G be a graph without negative weight cycles, that is without a cycle P where $w(P) < 0$. Then we can define the *shortest path* from v_0 to v_k, which we denote by $SP(v_0, v_k)$, as the (simple) path P from v_0 to v_k such that $w(P)$ is minimised.

Let $wSP(x, y) = w(SP(x, y))$ or $+\infty$ if no path exists from x to y.

Given a graph G and vertex x define the functions $\delta_x^{\leftarrow}, \delta_x^{\rightarrow} : V \to \mathbb{R}$ as

$$\delta_x^{\leftarrow}(y) = wSP(y, x) \qquad \text{and} \qquad \delta_x^{\rightarrow}(y) = wSP(x, y) \ .$$

Let G be a graph without negative weight cycles. Then π is a *valid potential*

function for G if $\pi(u) + d - \pi(v) \geq 0$ for every edge (u, v, d) in G. An edge (u, v, d) is called *tight* if $\pi(u) + d - \pi(v) = 0$.

There are many algorithms (see e.g. Cherkassky and Goldberg 1996) for detecting negative weight cycles in a weighted directed graph, which either detect a cycle or determine a valid potential function for the graph.

Given a valid potential function π for graph $G = (V, E)$ we can define the *reduced-cost graph* $rc(G)$ as $(V, \{(x, y, \pi(x) + d - \pi(y)) \mid (x, y, d) \in E\})$. All weights in the reduced cost graph are non-negative and we can recover the original path length $w(P)$ for path P from x to y from paths in the reduced cost graph since $w(P) = w + \pi(y) - \pi(x)$ where w is the weight of the corresponding path in the reduced-cost graph.

Since edges in the reduced-cost graph are non-negative we can use Dijkstra's algorithm (Dijkstra 1959) to calculate the shortest paths in the reduced-cost graph in time $\mathcal{O}(n \log n + m)$ instead of $\mathcal{O}(nm)$.

3.2.1 Difference Constraints

Difference constraints have the form $x - y \leq d$ where x and y are integer variables and $d \in \mathbb{Z}$. We can map difference constraints to a weighted directed graph.

Definition 3.1. Let C be a set of difference constraints and let $G = (V, E)$ be the graph comprised of one weighted edge $x \xrightarrow{d} y$ for every constraint $x - y \leq d$ in C. We call G the *constraint graph* of C.

The following well-known result characterises how the constraint graph can be used for satisfiability and implication checking of difference constraints.

Theorem 3.1. *Let C be a set of difference constraints and G its corresponding graph. C is satisfiable if and only if G has no negative weight cycles, and if C is satisfiable then $C \models x - y \leq d$ if and only if $wSP(x, y) \leq d$.* □

3.2.2 UTVPI Constraints

A UTVPI constraint is of the form $ax + by \leq d$, where x and y are integer variables, $a, b \in \{-1, 0, 1\}$ and $d \in \mathbb{Z}$.

Definition 3.2. The *transitive closure* $TC(\phi)$ of a set of UTVPI constraints ϕ is defined as the smallest set S containing ϕ such that

$$ax - y \leq d_1 \in S \wedge y + bz \leq d_2 \in S \quad \Rightarrow \quad ax + bz \leq d_1 + d_2 \in S$$

The *tightened closure* $TI(\phi)$ of a set of UTVPI constraints ϕ is defined as the smallest set S containing ϕ such that

$$ax + ax \leq d \in S \quad \Rightarrow \quad ax \leq \left\lfloor \frac{d}{2} \right\rfloor \in S, \qquad a \in \{-1, 1\}$$

The *tightened transitive closure* $TTC(\phi)$ of ϕ is the smallest set containing ϕ that satisfies both conditions.

The fundamental results for UTVPI constraints solving are (Jaffar *et al.* 1994):

Theorem 3.2 (Unsatisfiability, Jaffar *et al.* (1994))**.** *Let ϕ be a set of* UTVPI *constraints. Then ϕ is unsatisfiable if and only if exists $c \equiv (0 \leq d) \in TTC(\phi)$ where $d < 0$.* □

Theorem 3.3 (Implication, Jaffar *et al.* (1994))**.** *Let $c \equiv ax + by \leq d$ be a* UTVPI *constraint and let ϕ be a satisfiable set of* UTVPI *constraints. Then $\phi \models c$ if and only if either $c \equiv (0 \leq d)$ is a tautology, there exists $\{ax \leq d_1, by \leq d_2\} \subseteq TTC(\phi)$ with $d_1 + d_2 \leq d$, there exists $ax + by \leq d' \in TTC(\phi)$ with $d' \leq d$.* □

Example 3.1. Consider the UTVPI constraints $\phi \equiv \{x - y \leq 2,\ x + y \leq -1,\ -x - z \leq -4\}$, Then $TC(\phi)$ includes in addition $\{x + x \leq 1,\ -y - z \leq -2,\ y - z \leq -5, -z - z \leq -7, x - z \leq -3\}$. And $TI(TC(\phi))$ includes in addition $\{x \leq 0, -z \leq -4\}$ and $TTC(\phi) = TI(TC(\phi))$ in this case. The constraint $-z \leq -3$ is implied by ϕ as is $y - z \leq 0$. □

3.3 Lahiri and Musuvathi's Approach

Lahiri and Musuvathi map UTVPI constraints ϕ to difference constraints or equivalently a weighted directed graph G_ϕ, and they use graph algorithms to detect satisfiability.

We denote the constraint graph arising from ϕ as $G_\phi = (V, E)$. The graph G contains two vertices x^+ and x^- for every variable x. These variables are used to convert UTVPI constraints into difference constraints. The vertex x^+ represents $+x$ and x^- represents $-x$.

Let ϕ be a set of UTVPI constraints. Each UTVPI constraint $c \in \phi$ is mapped to a *set of difference constraints* $D(c)$, or equivalently a *set of weighted edges* $E(c)$. The mapping is shown in the Tab. 3.1. Each UTVPI constraint on two variables generates two difference constraints and accordingly two edges in the constraint graph. Each UTVPI constraint on a single variable generates a single constraint, and hence a single edge.

Table 3.1: Transformation from UTVPI constraint c to associated difference constraints $D(c)$ to edges in the constraint graph $E(c)$.

UTVPI c	Diff. Constr. $D(c)$	Edges $E(c)$
$x - y \leq d$	$x^+ - y^+ \leq d$ $y^- - x^- \leq d$	$x^+ \xrightarrow{d} y^+$ $y^- \xrightarrow{d} x^-$
$x + y \leq d$	$x^+ - y^- \leq d$ $y^+ - x^- \leq d$	$x^+ \xrightarrow{d} y^-$ $y^+ \xrightarrow{d} x^-$
$-x - y \leq d$	$x^- - y^+ \leq d$ $y^- - x^+ \leq d$	$x^- \xrightarrow{d} y^+$ $y^- \xrightarrow{d} x^+$
$x \leq d$	$x^+ - x^- \leq 2d$	$x^+ \xrightarrow{2d} x^-$
$-x \leq d$	$x^- - x^+ \leq 2d$	$x^- \xrightarrow{2d} x^+$

Let $-u$ denote the counterpart of a vertex $u \in V$, i.e. $-x^+ := x^-$ and $-x^- := x^+$. Clearly, for each edge $(u, v, d) \in E$ the graph G_ϕ also includes the edge $(-v, -u, d)$ (called its *counteredge*) which has equal weight. This correspondence extends to paths.

Lemma 3.1 (Lahiri and Musuvathi (2005))**.** *If there is a path P from u to v in the constraint graph G_ϕ, then there is a* counterpath *path P' from $-v$ to $-u$ such that $w(P) = w(P')$.* □

If we relax the restriction on variables to take values in $\mathbb{Z}$ and allow them to take values in $\mathbb{Q}$ we can check satisfiability in $\mathbb{Q}$ using G_ϕ.

Lemma 3.2 (Lahiri and Musuvathi (2005))**.** *A set of* UTVPI *constraints ϕ is unsatisfiable in $\mathbb{Q}$ if and only if the constraint graph $G_\phi = (V, E)$ contains a negative weight cycle.* □

The reason why the satisfiability in $\mathbb{Z}$ cannot be tested with G_ϕ arises from the possible implication of constraints of the form $x + x \leq d$ or $-x - x \leq d$ through the transitivity of constraints in ϕ. If d is odd (equivalently $d/2 \in \mathbb{Q} \setminus \mathbb{Z}$) then ϕ may be satisfiable with $x = d/2$ but not with $x = \lfloor d/2 \rfloor$.

Example 3.2. Consider the UTVPI problem $\phi' \equiv \{x - y \leq 2, x + y \leq -1, -x - z \leq -4, -x + z \leq 3\}$, then a transitive consequence of the first two is $x + x \leq 1$, while a consequence of the second two is $-x - x \leq -1$. Together these require $x = \frac{1}{2}$.

The graph $G_{\phi'}$ is shown in Fig. 3.1(a). A zero weight cycle is extracted in Fig. 3.1(b). This cycle has solutions in $\mathbb{Q}$ but not in $\mathbb{Z}$. □

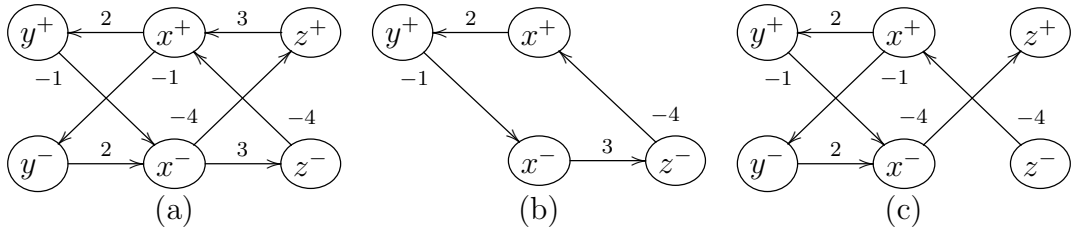


Figure 3.1: (a) $G_{\phi'}$ for ϕ' of Example 3.2 which is $\mathbb{Q}$-satisfiable but not $\mathbb{Z}$ satisfiable. (b) a zero weight cycle in $G_{\phi'}$. (c) G_ϕ for ϕ of Example 3.1.

Algorithm 3.1: LaMu (Lahiri and Musuvathi 2005)

Input: ϕ a set of UTVPI constraints
Output: Satis if ϕ is satisfiable, Unsat otherwise

1 Construct the constraint graph $G_\phi = (V, E)$ from ϕ;
2 Run a negative cycle detection algorithm on G_ϕ;
3 **if** *G_ϕ contains a negative cycle* **then**
4 **return** *Unsat*
5 **else**
6 let π be a valid potential function for G_ϕ
7 $E' := \{(u, v) \mid (u, v, d) \in E, \pi(u) + d = \pi(v)\}$;
8 $H_\phi := (V, E')$;
9 Group the vertices in H_ϕ into strongly connected components (SCCs). Vertices u and v are in the same SCC if and only if there is a path from u to v and a path from v to u in H_ϕ. u and v are in the same SCC exactly when there is a zero weight cycle in G_ϕ containing u and v;
10 **for all** $u \in V$ **do**
11 **if** *$-u$ is in the same* SCC *as u in H_ϕ* **and** *$\pi(-u) - \pi(u)$ is odd* **then**
12 **return** *Unsat*
13 **return** *Satis*

The satisfaction algorithm of Lahiri and Musuvathi (2005) is based on Lem. 3.2 and the following result.

Lemma 3.3 (Lahiri and Musuvathi (2005))**.** *Suppose G_ϕ has no negative cycles and ϕ is unsatisfiable in $\mathbb{Z}$. Then G_ϕ contains a zero weight cycle containing vertices u and $-u$ such that $wSP(u, -u)$ is odd.* □

The Lahiri and Musuvathi algorithm is shown in Alg. 3.1: LaMu. The algorithm first checks $\mathbb{Q}$ satisfiability using a negative cycle detection algorithm (line 3), and then checks that no such zero weight cycles exists in G_ϕ (lines 5–13) while building up an auxiliary graph H_ϕ containing all tight edges E' and determining all its strongly connected components (Scc).

Example 3.3. A valid potential function for the graph shown in Fig. 3.1(a) is

$\pi(y^+) = 5$, $\pi(x^+) = 3$, $\pi(z^+) = 0$, $\pi(y^-) = 2$, $\pi(x^-) = 4$, $\pi(z^-) = 7$. Each of the edges is tight, so E' contains all edges, and all nodes are in the same Scc. Both x^+ and x^- occur in the same Scc and $SP(x^+, x^-) = \pi(x^-) - \pi(x^+) = 1$ is odd, hence the system is unsatisfiable. □

The complexity of LaMu is $\mathcal{O}(nm)$ time and $\mathcal{O}(n + m)$ space assuming the application of the Bellman-Ford single source shortest path algorithm (Bellman 1958, Ford and Fulkerson 1962) for negative cycle detection.

3.4 Incremental UTVPI Satisfaction

The incremental satisfaction problem is: Given a satisfiable set of Utvpi ϕ (with n variables and m constraints) and Utvpi constraint c, determine if $\phi \cup \{c\}$ is satisfiable. In this section we define an incremental satisfaction checker for Utvpi constraints that requires $\mathcal{O}(n \log n + m)$ time and $\mathcal{O}(n + m)$ space. It relies on simply making incremental the algorithm LaMu of Lahiri and Musuvathi. We examine the two major components of their algorithm: negative cycle detection, and the calculation of strongly connected components (Sccs).

The key is to make the negative cycle detection and the Scc computation incremental. We make use of incremental negative cycle detection algorithms previously used for difference constraints. We also carefully consider how the Sccs can change under the addition of constraints, in order to minimise the work in recalculating Sccs.

For incremental negative cycle detection we use an algorithm due to Frigioni *et al.* (1998), using the simplified form (Alg. 3.2: IncConDiff) of Cotton and Maler (2006) (since we are not interested in edge deletion). Given a graph $G = (V, E)$ and valid potential function π for G and edge $e = u \xrightarrow{d} v$, this algorithm returns $G' = (V, E \cup \{e\})$ and a valid potential function π' for G' or determines a negative cycle and returns Unsat. The complexity is $\mathcal{O}(n \log n + m)$ time and $\mathcal{O}(n + m)$ space using Fibonacci heaps to implement argmin.

IncConDiff works as follow: it takes a copy π' of the valid potential function π and the edge to be added (u, v, d) and repairs all π' values of nodes t violating $\pi'(s) + d' - \pi'(t) \leq 0$ for an edge $(s, t, d') \in G_{\phi \cup \{u - v \leq d\}}$. If the potential function is valid after the addition of (u, v, d), *i.e.*, $\gamma(v) \geq 0$ (line 2) then the conditions (line 4 and 11) are not satisfied and algorithm terminates straightforwardly (line 13). Otherwise the π' value of v is fixed at first by adding $\gamma(v)$ to its previous value $\pi(v)$. This can lead to violation of the potential function condition on its outgoing edges (v, t). For all these edges the node t is added to the priority queue with priority $\gamma(t)$

Algorithm 3.2: IncConDiff (Cotton and Maler 2006)

Input: $G_\phi = (V, E)$ a graph, π a valid potential function for G_ϕ, edge (u, v, d) a new constraint to add to G_ϕ.

Output: Unsat if $\phi \cup \{u - v \leq d\}$ is unsatisfiable, or $G_{\phi \cup \{u-v \leq d\}}$ and a valid potential function π' for $G_{\phi \cup \{u-v \leq d\}}$.

```
1  π'(v) := π(u) + d − π(v);
2  γ(v) := π(u) + d − π(v);
3  γ(w) := 0 for all w ≠ v;
4  while min(γ) < 0 ∧ γ(u) = 0 do
5      s := argmin(γ) ;
6      π'(s) := π(s) + γ(s) ;
7      γ(s) := 0 ;
8      for all s --d'--> t ∈ G do
9          if π'(t) = π(t) then
10             γ(t) := min{γ(t), π'(s) + d' − π'(t)}
11 if γ(u) < 0 then
12     return Unsat
13 return ((V, E ∪ {(u, v, d)}), π')
```

(line 10). The same procedure is applied for fixing the π' values of all nodes in the queue starting with the node with a lowest priority. The algorithm proves either satisfiability if it can empty the queue or unsatisfiability if $\gamma(u)$ becomes negative (line 10). Implementation details can be found in Cotton and Maler (2006).

To make the Scc computation of LaMu in the zero reduced-cost graph incremental, we use the same Scc algorithm as before, but restrict its application. The *zero reduced-cost graph* is the subgraph of the reduced-cost graph with all zero weight edges. The key to incrementalising the Scc calculation is the following lemma which shows that the addition of a new constraint will either make no change to the Sccs in a manner which is quick to detect, or can only collapse Sccs for the graph of the previous problem.

Lemma 3.4. *Let ϕ be a satisfiable set of* Utvpi *constraints and π_ϕ a valid potential function for G_ϕ. Let H_ϕ be the graph of zero weight reduced-cost edges in G_ϕ. Let $G_{\phi'}$ be G_ϕ with the addition of an edge (u, v, d) and define analogously $\pi_{\phi'}$ and $H_{\phi'}$. If $\pi_\phi(u) + d > \pi_\phi(v)$ the* Sccs *of H_ϕ and $H_{\phi'}$ are identical. If $\pi_\phi(u) + d \leq \pi_\phi(v)$ then the* Sccs *of $H_{\phi'}$ are either identical to those of H_ϕ or result from the union of* Sccs *in H_ϕ reachable from v in G_ϕ.*

Proof. We carry out the proof with respect to IncConDiff. If $\pi_\phi(u) + d > \pi_\phi(v)$ then $\gamma(v) > 0$ and no potential function values change, so $H_{\phi'} = H_\phi$ and the result

holds.

If $\pi_\phi(u) + d = \pi_\phi(v)$ then $\gamma(v) = 0$ and no potential function values change, so $H_{\phi'} = H_\phi$ with the addition of the edge (u, v) if not already present. Clearly, this can only result in the union of SCCs of H_ϕ reachable from v.

If $\pi_\phi(u) + d < \pi_\phi(v)$ then $\gamma(v) < 0$ and IncConDiff does create a new potential function $\pi_{\phi'}$. We show that each node s reachable from v in H_ϕ has its potential function value changed to $\pi_{\phi'}(s) = \pi_\phi(s) + \gamma(v)$ by induction on path length from v.

The base case clearly holds.

Suppose the result holds for s reachable from v in k steps. Each node t where $(s, t) \in H_\phi$ is such that there exists an edge (s, t, d') in G_ϕ where $\pi_\phi(s)+d'-\pi_\phi(t) = 0$. When s is selected for updating in IncConDiff, then each t which has not been changed already, has $\gamma(t)$ set to $\gamma(v)$. When t is selected IncConDiff will define $\pi_{\phi'}(t) = \pi_\phi(t) + \gamma(v)$.

Clearly then each edge $(s, t) \in H_\phi$ also exists in $H_{\phi'}$.

Thus SCCs in $H_{\phi'}$ must be unions of SCCs in H_ϕ, and since the only potential values changing are those reachable from v in G_ϕ the result holds. □

The incremental Utvpi satisfaction algorithm Satis (shown in Alg. 3.3) simply runs IncConDiff at line 3 of LaMu. If the newly added edge (u, v, d) has a reduced cost of zero it has to rebuild the zero reduced-cost graph H_ϕ graph to determine possible $\mathbb{Z}$ unsatisfiability, since the SCCs of the zero reduced-cost graph may have changed. It does so only for nodes reachable from v in G_ϕ. Therefore Satis computes the SCCs of the subgraph H (defined at line 8) just containing these nodes and their incident zero reduced-cost edges. Then it uses the same check as LaMu for $\mathbb{Z}$ unsatisfiability (line 10). The algorithm requires $\mathcal{O}(n + m)$ time and space for the Scc construction and checking, and hence the overall complexity is dominated by IncConDiff requiring $\mathcal{O}(n \log n + m)$ time and $\mathcal{O}(n + m)$ space.

Theorem 3.4. *Algorithm 3.3 (*Satis*) is correct and runs in $\mathcal{O}(n \log n + m)$ time and $\mathcal{O}(n + m)$ space.*

Proof. The complexity follows straightforwardly, recall that constructing SCCs requires $\mathcal{O}(n + m)$ time and space. The correctness follows from the correctness of LaMu (Lem. 3.2 and 3.3) as well as Lem. 3.4 which allows us to avoid generating the SCCs of the entire graph $G_{\phi \cup \{c\}}$. □

Algorithm 3.3: SATIS – Incremental satisfaction for UTPVI constraints.

Input: ϕ a satisfiable set of UTVPI constraints, $G_\phi = (V, E)$ is the constraint graph of ϕ, π is a valid potential function for G_ϕ, c a UTVPI constraint

Output: Satis if $\phi \cup \{c\}$ is satisfiable, Unsat otherwise

```
1  for all (u, v, d) ∈ E(c) do
2      r := IncConDiff((V, E),π,(u, v, d));
3      if r = Unsat then
4          return Unsat
5      else
6          ((V, E), π) := r
7      if π(u) + d − π(v) = 0 then
8          Determine the SCCs of the graph
           H = (V, {(s, t) | (s, t, d′) ∈ E, π(x) + d′ = π(y)}) reachable from the
           node v;
9          for all s ∈ V reachable from v in H do
10             if −s is in the same SCC as s in H and π(−s) − π(s) is odd then
11                 return Unsat
12 return Satis
```

3.5 Incremental UTVPI Implication

The incremental implication problem is given by a set P of $p = |P|$ UTVPI constraints and a satisfiable set ϕ of m UTVPI constraints on n variables, where $\phi \not\models c', \forall c' \in P$, as well as a single new UTVPI constraint c. For each $c' \in P$ the incremental implication problem is to check if $\phi \wedge c \models c'$.

Incremental implication is important if we wish to use UTVPI constraints in an SMT solver, as well as for uses in abstract interpretation and spatial databases. Our approach to incremental implication is similar to the approach of Cotton and Maler (2006) for incremental implication for difference constraints. The fundamental new insight that our implication algorithm exploits is that building the tightened transitive closure TTC of a constraint set ϕ can be managed by first building the transitive closure $TC(\phi)$ and then applying the tightening on it.

In this section we first prove that $TTC(\phi) = TI(TC(\phi))$ using two lemmas. This means that we can determine most of the information about the tightened transitive closure by considering transitive closure from the constraint graph. We then show how, using this insight, we can reason about UTVPI implication simply using shortest paths, as well as a function to extract the upper and lower bounds of variables directly from the constraint graph.

To prove some of the following results we introduce the notion of a *proof* of a constraint being a member of $TTC(\phi)$ as follows:

Definition 3.3. A *proof* of a constraint $c \in TTC(\phi)$ (or analogously $TI(\phi)$ or $TC(\phi)$) is a tree of nodes labelled by constraints. The root of the tree is labelled c. If the constraint c is generated by transitive closure of c_1 and c_2 then a node labelled c has two child nodes labelled c_1 and c_2. If the constraint c is generated by tightening of c' then the node labelled c has a single child node labelled c'. If $c \in \phi$ then c is a leaf.

The next two lemmas show that $TTC(\phi)$ can be built up by the two closure steps $TI(TC(\phi))$. The first show that a constraint involving two variables in the tightened transitive closure is created from the transitive closure operator, *i.e.*, in other words a tightening introduces constraints involving a single variable and any further transitive closure involving them can only create new constraints involving a single variable.

Lemma 3.5. *Let* $ax + by \leq d \in TTC(\phi)$ *where* $\{a, b\} \subseteq \{-1, 1\}$ *then* $ax + by \leq d \in TC(\phi)$.

Proof. Define a constraint $ax + by \leq d$ as a binary constraint if $\{a, b\} \subseteq \{-1, 1\}$.

The proof is by induction on proof size. If $c \in TTC(\phi)$ then it has a finite proof. If the proof size is 0, then c is a leaf and $c \in TC(\phi)$.

Suppose the result holds for all proofs of size less than k. Let $c \in TTC(\phi)$ be a binary constraint with proof size k. Now c is generated using the transitive closure rule, since the tightening rule cannot generate binary constraints. Examining the transitive closure rule, if the result is binary then the generating constraints $c_1 \equiv ax - y \leq d_1$ and $c_2 \equiv y + bz \leq d_2$ are also binary. Since the proof for the constraints c_1 and c_2 are less than k then by induction, $\{c_1, c_2\} \subseteq TC(\phi)$, hence $c \in TC(\phi)$. □

For $TTC(\phi) = TI(TC(\phi))$ we only have to show that any result of transitive closure on a new UTVPI constraint $by \leq d'$ introduced by tightening, can be mimicked using the constraint $by + by \leq \{2d', 2d' + 1\}$ that introduced it, and tightening the end result.

Lemma 3.6. *Let* $ax \leq d \in TTC(\phi)$ *where* $a \in \{-1, 1\}$ *then* $ax \leq d \in TI(TC(\phi))$.

Proof. We show that for each $ax \leq d \in TTC(\phi)$ either $ax \leq d \in TC(\phi)$ or $ax + ax \leq d' \in TC(\phi)$ where $d' = 2d$ or $d' = 2d + 1$. Then clearly $ax \leq d \in TI(TC(\phi))$.

The proof is by induction on proof size. Let $c \in TTC(\phi)$. If the proof size is 0 then $c \in \phi$ and $c \in TI(TC(\phi))$. Suppose the result holds for all proofs of size less than k. Let $ax \leq d \in TTC(\phi)$ with proof size k.

If the root node is a transitive closure node with children labelled c_1 and c_2 then exactly one of c_1 and c_2 is binary. Assume c_1 is binary, the other case is similar. Now $c_1 \equiv ax - y \leq d_1$ and $c_2 \equiv y \leq d_2$ and $d = d_1 + d_2$. By Lem. 3.5 $c_1 \in TC(\phi)$. Since the proof of c_2 has size less than k by induction we have that $y \leq d_2 \in TC(\phi)$ or $y + y \leq d_2 + d_2 + e \in TC(\phi)$—if $y \leq d_2 \notin TC(\phi)$—where $e \in \{0, 1\}$. In the first case, clearly $ax \leq d \in TC(\phi)$. In the second case, we have $ax + ax \leq d_1 + d_1 + d_2 + d_2 + e \in TC(\phi)$ by two applications of the transitive closure on c_1 and c_2, and then on c_1 again. Hence, the induction hypothesis holds.

If the root node is a tightening node with child c' then c' is binary and hence by Lem. 3.5 $c' \in TC(\phi)$, and the induction hypothesis holds, too. □

The above two results show that $TC(\phi)$ is the crucial set of interest for UTVPI implication checking. The following result shows how the constraint graph can be used to reason about $TC(\phi)$. It also shows implicitly (in combination with Lem. 3.2) that the satisfaction in $\mathbb{Q}$ is decided by $TC(\phi)$ if a constraint $0 \leq d \in TC(\phi)$ where d is negative.

Lemma 3.7. *$c \in TC(\phi)$ if and only if there is a cycle of length d, in the case of $c \equiv 0 \leq d$, or a path $u \rightsquigarrow v$ of length d in G_ϕ where $(u, v, d) \in E(c)$.*

Proof. ($\Rightarrow$): The proof is by induction on proof size. Clearly if the proof size is 0, then $c \in TC(\phi)$ and $E(c)$ appear in the graph G_ϕ. Let $c \equiv ax + bz \leq d_1 + d_2$ have proof size k. Then c is built using $c_1 \equiv ax - y \leq d_1$ and $c_2 \equiv y + bz \leq d_2$. Assume for simplicity $a = 1$, and $b = -1$. The remaining cases are similar.

By induction there exists a path from $x^+ \rightsquigarrow y^+$ of length d_1 in G_ϕ and there exists a path $y^+ \rightsquigarrow z^+$ of length d_2 in G_ϕ. Hence there is a path of length $d_1 + d_2$ from x^+ to z^+. Now if $x = z$ this is a cycle.

($\Leftarrow$): The proof is by induction on the number of edges in path $u \rightsquigarrow v$. If the number of edges is 1 then $(u, v, d) \in G_\phi$ and hence $c \in \phi$.

Let $u \rightsquigarrow v$ be a path of length d involving k edges, then it has the form $u \rightsquigarrow w$ of length d_1 and $(w, v, d_2) \in G_\phi$ where $d = d_1 + d_2$. Now by induction there is $(u, w, d_1) \in E(c_1)$ for some $c_1 \in TC(\phi)$ and $(w, v, d_2) \in E(c_2)$ for some $c_2 \in \phi$. Assume c_1 is of the form $x - y \leq d_1$, and c_2 is of the form $y - z \leq d_2$, where $w = y^+$, $u = x^+$ and $v = z^+$. The other cases are similar. Then by transitive closure $c \equiv x - z \leq d_1 + d_2 \in TC(\phi)$ and $(u, v, d) \in E(c)$. □

Example 3.4. Consider ϕ of Ex. 3.1. Then for example $x + x \leq 1 \in TC(\phi)$ and there is a path $x^+ \rightsquigarrow x^-$ of length 1 in G_ϕ shown in Fig. 3.1(c). Similarly $y - z \leq -5 \in TC(\phi)$ and there are paths $z^- \rightsquigarrow y^-$ and $y^+ \rightsquigarrow z^+$ of length -5 in G_ϕ. □

The consequence of Lem. 3.7 is that we can use paths not only to reason about all constraints in $TC(\phi)$ but also to infer about the tightened constraints in $TTC(\phi)$ by looking for paths from a node u to its counternode $-u$ in the constraint graph G_ϕ. That means no tightened edges need to be added to G_ϕ just as in the satisfiability case. But still tightening has to be handled. For that we introduce a *bounds function* ρ which records the upper and lower bounds for each variable x, on the vertices x^+ and x^-. It is defined as:

$$\rho(u) = \left\lfloor \frac{wSP(u, -u)}{2} \right\rfloor .$$

We can show that $\rho(x^+)$ is the upper bound of x and $-\rho(x^-)$ is the lower bound of x. Using Lem. 3.6 and 3.7 we have.

Lemma 3.8. *For* Utvpi *constraints* ϕ*,*

$$\rho(x^+) = \min\{d \mid x \leq d \in TTC(\phi)\}$$
$$\rho(x^-) = \min\{d \mid -x \leq d \in TTC(\phi)\}$$

where we assume $\min \emptyset = +\infty$. □

Example 3.5. Consider the graph in Fig. 3.1(c) for constraints ϕ of Ex. 3.1. Then $\rho(x^+) = 0$ since $wSP(x^+, x^-)$ equals 1 and $x \leq 0 \in TTC(\phi)$, while $\rho(z^-) = -4$ since $wSP(z^-, z^+) = -7$ and $-z \leq -4 \in TTC(\phi)$. Note e.g. $\rho(x^-) = +\infty$ and there is no constraint of the form $-x \leq d$ in $TTC(\phi)$. □

The next two theorems state how the constraint graph G_ϕ and the bounds function ρ can be used to reason about satisfaction and implication. The key to incremental satisfaction is the following result.

Theorem 3.5. *If the constraint graph* G_ϕ *contains no negative weight cycle (i.e.* ϕ *is satisfiable in* $\mathbb{Q}$*) then* ϕ *is unsatisfiable in* $\mathbb{Z}$ *if and only if a vertex* $v \in V$ *exists with* $\rho(v) + \rho(-v) < 0$*.*

Proof. Let ϕ be a satisfiable set of Utvpi constraints in $\mathbb{Q}$. By Lem. 3.7 there is no constraint $0 < d \in TC(\phi)$ where $d < 0$. Therefore ϕ is unsatisfiable in $\mathbb{Z}$ if and only if such a constraint belongs to $TTC(\phi) \setminus TC(\phi)$ (Thm. 3.2), i.e. a possible unsatisfiability is caused by tightening.

Lemma 3.5 implies the equivalence for each constraint $c \in TTC(\phi) \setminus TC(\phi)$ to $ax \leq d$ where $a \in \{-1, 0, 1\}$. Hence, ϕ is unsatisfiable in $\mathbb{Z}$ if and only if two constraints $x \leq d_1$ and $-x \leq d_2$ with $d_1 + d_2 < 0$ exist in $TTC(\phi)$ if and only if (by Lem. 3.8) $\rho(x^+) + \rho(x^-) < 0$. □

Effectively failure can only be caused by tightening if the bounds of a single variable contradict.

Example 3.6. Consider the graph in Fig. 3.1(a) for constraints ϕ' of Ex. 3.2. There is no negative weight cycle in $G_{\phi'}$ but $\rho(x^+) = 0$ and $\rho(x^-) = -1$ because of $x^- \xrightarrow{-4} z^+ \xrightarrow{3} x^+$. Hence the system is unsatisfiable. □

Similarly the key to incremental implication is the following rephrasing of Thm. 3.3.

Theorem 3.6. *If ϕ is a satisfiable set of* UTVPI *constraints then $\phi \models c$ if and only if for at least one $(u, v, d) \in E(c)$ either $wSP(u, v) \leq d$ or $\rho(u) + \rho(-v) \leq d$.*

Proof. Let ϕ be a satisfiable set of UTVPI constraints. Because of Thm. 3.3 it holds $\phi \models c$ and $c \equiv ax + by \leq d$ if and only if $ax + by \leq d' \in TTC(\phi)$ and $d' \leq d$ or $\{ax \leq d_1, by \leq d_2\} \subseteq TTC(\phi)$ and $d_1 + d_2 \leq d$.

Now, the theorem holds straightforwardly due to Lem. 3.8 for the constraints with one variable, and Lem. 3.5 and 3.7 for the other constraints. □

Example 3.7. Consider the graph in Fig. 3.1(c) for constraints ϕ of Ex. 3.1. $\phi \models -z \leq -3$ is shown since $wSP(z^-, z^+) = -7 \leq 2 \times -3$. □

Algorithm 3.4: IMPL shows the new algorithm. As input it takes the constraint graph G_ϕ, a valid potential function π, the bounds function ρ, a set P of UTVPI constraints to check for implication, as well as the UTVPI constraint c which should be added to ϕ.

In the first step (line 1) the constraint c is transformed to its corresponding edges $E(c)$ in a constraint graph. Then each edge in $E(c)$ is added consecutively to the constraint graph G_ϕ by using the INCCONDIFF algorithm of Cotton and Maler (2006). After inserting all edges in G', the constraint graph equals $G_{\phi \cup \{c\}}$ and π' is a valid potential function for G'. Hence $\phi \cup \{c\}$ is satisfiable in $\mathbb{Q}$. The remainder of the algorithm maintains the bounds function ρ' (lines from 8 to 12) and it is used to test the satisfiability in $\mathbb{Z}$ (lines 13 and 14), and the implication of constraints in P (lines 15 to 18). For building an efficient implementation of shortest paths the reader is referred to Cotton and Maler (2006).

By Lem. 3.7 to maintain ρ we need to see if the shortest path from x to $-x$ has changed. We only need to scan for new shortest paths using the newly added

Algorithm 3.4: IMPL – Incremental satisfaction and implication for UTVPI constraints.

Input: $G_\phi = (V, E)$ a *constraint graph* representing set of UTVPI constraints ϕ, π a *valid potential function* on G_ϕ, ρ the *bounds function* of ϕ, P a set of UTVPI constraints not implied by ϕ, and a UTVPI constraint c to be added.

Output: $G_{\phi\cup\{c\}}$, a *valid potential function* π' and the *bounds function* ρ' of $\phi \cup \{c\}$ and the set $P' \subseteq P$ of constraints not implied by $\phi \cup \{c\}$, or Unsat if $\phi \cup \{c\}$ is not satisfiable.

1 $G' := G_\phi$, $\pi' := \pi$, $\rho' = \rho$, compute $E(c)$;
2 **for all** $e \in E(c)$ **do**
3 $res :=$ INCCONDIFF(G', π', e);
4 **if** $res =$ *Unsat* **then**
5 **return** *Unsat*
6 **else**
7 $(G', \pi') := res$
8 let (u, v, d) be any edge in $E(c)$;
9 compute $\delta_u^{\leftarrow}$ and $\delta_v^{\rightarrow}$ by using the reduced-cost graph for G' via π';
10 **for all** $x \in V$ **do**
11 $sp := \delta_u^{\leftarrow}(x) + d + \delta_v^{\rightarrow}(-x)$;
12 $\rho'(x) := \min\{\rho(x), \lfloor \frac{sp}{2} \rfloor\}$;
13 **for all** $x \in V$ **do**
14 **if** $\rho'(x) + \rho'(-x) < 0$ **then return** *Unsat*;
15 $P' := \emptyset$;
16 **for all** $c' \in P$ **do**
17 $(x, y, d') :=$ first element in $E(c')$;
18 **if** $\delta_u^{\leftarrow}(x) + d + \delta_v^{\rightarrow}(y) > d'$ **and** $\delta_u^{\leftarrow}(-y) + d + \delta_v^{\rightarrow}(-x) > d'$ **and** $\rho'(x) + \rho'(-y) > d'$ **then** $P' := P' \cup \{c'\}$;
19 **return** *(G', π', ρ', P')*

edges. We can restrict attention to a single added edge (u, v, d) since if there is a path from x over the edge (u, v, d) to $-x$ ($x^+ \rightsquigarrow u \xrightarrow{d} v \rightsquigarrow x^-$) then because of Lem. 3.1 there is equal-weight path from x via the "counter-edge" $(-v, -u, d)$ to $-x$ ($x^+ \equiv -x^- \rightsquigarrow -v \xrightarrow{d} -u \rightsquigarrow -x^+ \equiv x^-$).

We calculate the shortest paths in $G_{\phi\cup\{c\}}$ from each vertex x to u ($\delta_u^{\leftarrow}(x)$) and from v to each vertex x ($\delta_v^{\rightarrow}(x)$) (line 9). The shortest path for $\delta_u^{\leftarrow}$ can be computed like $\delta_u^{\rightarrow}$ by simply reversing the edges in the graph.

We can then calculate the shortest path from x to $-x$ via the edge $u \xrightarrow{d} v$ using the path $x^+ \rightsquigarrow u \xrightarrow{d} v \rightsquigarrow x^-$ as $\delta_u^{\leftarrow}(x) + d + \delta_v^{\rightarrow}(-x)$. We update ρ' if required (line 12).

We can now check satisfiability of $\phi \cup \{c\}$ in $\mathbb{Z}$ using Thm. 3.5 (lines 13 and 14).

Finally we check implications using Thm. 3.6.

Using the above results, it is not difficult to show that the algorithm is correct with the desired complexity bounds.

Theorem 3.7. *Algorithm 3.4 (*Impl*) is correct and runs in* $\mathcal{O}(n \log n + m + p)$ *time and* $\mathcal{O}(n + m + p)$ *space.*

Proof. The algorithm is correct if it returns Unsat in the case of unsatisfiability of $\phi \cup \{c\}$ or the constraint graph $G_{\phi \cup \{c\}}$, its valid potential function π', its bounds function ρ' and the set of constraints $P' \subseteq P$ not implied by $\phi \cup \{c\}$.

Lemma 3.2 and algorithm IncConDiff (see Cotton and Maler (2006)) guarantee that after termination of IncConDiff $G' = G_{\phi \cup \{c\}}$ and π' is its valid potential function if $\phi \cup \{c\}$ is satisfiable in $\mathbb{Q}$; otherwise $\phi \cup \{c\}$ is unsatisfiable and the algorithm returns Unsat.

After application of IncConDiff the algorithm maintains the bounds function (lines 8 to 12) by calculation of the shortest path $x \rightsquigarrow u \to v \rightsquigarrow -x$ via one added edge $(u, v, d) \in E(c)$ for each node x in $G_{\phi \cup \{c\}}$. Remark that we only have to consider the shortest paths via the added edges, since ρ give us the length of a shortest path without those added edges. Due to Thm. 3.5 the algorithm checks $\phi \cup \{c\}$ for unsatisfiability in $\mathbb{Z}$ in the next two lines. If it is unsatisfiable Impl terminates and returns Unsat.

The remainder of the algorithm computes the set of non-implied constraints $P' \subseteq P$ by testing for all constraints $c' \in P$ if the length of both paths $x \rightsquigarrow u \to v \rightsquigarrow y$, $-y \rightsquigarrow u \to v \rightsquigarrow -x$ are longer than d' and the sum of the upper bounds $\rho'(x) + \rho'(-y)$ is greater than d' where $(x, y, d') \in E(c')$. If all three cases hold then c' is not implied by $\phi \cup \{c\}$ thanks to Thm. 3.6.

The run-time is determined by the run-time of IncConDiff, the calculation of $\delta_u^{\leftarrow}$, $\delta_v^{\rightarrow}$ which are $\mathcal{O}(n \log n + m)$, and the implication check $\mathcal{O}(p)$. All the other computations can be done in constant or linear time with respect to n and m. So the overall run-time is $\mathcal{O}(n \log n + m + p)$.

The space is determined by the space to store the graph and implication constraints so it is $\mathcal{O}(n + m + p)$. The algorithm only needs to attach a constant amount of information to parts of the graph. □

3.6 Experimental Results

We present empirical comparisons of the algorithms discussed herein, first on satisfaction and then on implication questions.

For both experiments we generate 60 UTVPI instances ϕ in each problem class with the following specifications: the values d range uniformly in from -15 to 100, approximately 10% are negative, each variable appears in at least one UTVPI constraint, each constraint involves exactly two variables, and there is at most one constraint allowed between any two variables.

The experiments were run on a Sun Fire T2000 running SunOS 5.10 and a 1 GHz processor. The code was written in C and compiled with gcc 3.4.3 and option -O3. Each run was given a 2 minute time limit.

We run incremental satisfaction on a system of m constraints in n variables, adding the constraints one at a time. We compare: SATIS the incrementalisation of LAMU presented in Sec. 3.4, IMPL the incremental implication checking algorithm of Sec. 3.5 where $p = 0$, mLAMU running LAMU m times for m satisfaction checks, and HAST the algorithm of Harvey and Stuckey (1997). For the computation of the shortest paths we used a binary priority queue for argmin, and the early termination and caliber heuristics (Cotton and Maler 2006).

The results are shown in left of Tab. 3.2, where d represents the density of a UTVPI instance, which is defined as $m/2n^2$ representing the percentage of the number of constraints m in the instance to the maximal number of non-quasi-syntactic redundant constraints for n variables. A constraint $ax + by \leq d$ is *quasi-redundant* with respect to ϕ if and only if $ax + by \leq d' \in \phi$ with $d' < d$ applies. We split the examples into cases that are satisfiable, $\mathbb{Z}$ unsatisfiable, and $\mathbb{Q}$ unsatisfiable. Moreover, the fourth row shows the number of examples for each case written (satisfiable, $\mathbb{Z}$ unsatisfiable, $\mathbb{Q}$ unsatisfiable) and the overall average runtime. For more dense satisfiable systems IMPL is best, but overall SATIS is the clear winner. Interestingly for very dense satisfiable systems (not shown here) HAST beats the others.

For the implication benchmarks we chose 5 satisfiable, $\mathbb{Z}$ unsatisfiable, and $\mathbb{Q}$ unsatisfiable instances for each problem class. In addition, 5 implication sets P of size p were created for each n using the same restrictions as defined above. On average over all benchmarks, 65% of the constraints in P were implied by the corresponding ϕ.

The incremental implication experiments check satisfiability and the implications of constraints P incrementally as each of the m constraints were added one at a time. A run was terminated if there were no more constraints to add, or unsatisfiability was detected. We compare the two algorithms that can check implication: IMPL versus HAST. The right of Tab. 3.3 shows the results. Overall the checks for implication are cheap compared to the satisfaction check for HAST, but not for IMPL, since it must compute the shortest path after each constraint addition. Hence the times

Table 3.2: Average runtime in seconds of the satisfiability algorithms

Examples		Satis	Impl	*m*LaMu	HaSt
$n = 100$	feasible	0.03	0.13	1.02	1.56
$m = 1000$	Z-inf.	0.01	0.09	0.57	1.42
$d = 5\%$	Q-inf.	0.01	0.04	0.26	0.81
all (32, 8, 20)		0.02	0.10	0.70	1.29
$n = 100$	feasible	0.18	0.34	3.81	1.98
$m = 2000$	Z-inf.	<0.01	0.14	1.25	1.68
$d = 10\%$	Q-inf.	0.02	0.03	0.21	0.68
all (31, 9, 20)		0.10	0.20	2.23	1.50
$n = 100$	feasible	0.98	0.94	12.39	2.31
$m = 4000$	Z-inf.	0.02	0.15	1.17	1.82
$d = 20\%$	Q-inf.	0.01	0.04	0.25	0.80
all (28, 12, 20)		0.46	0.48	6.10	1.71
$n = 200$	feasible	0.73	1.28	16.84	16.72
$m = 4000$	Z-inf.	0.03	0.43	3.17	13.34
$d = 5\%$	Q-inf.	0.01	0.11	0.91	6.01
all (30, 11, 19)		0.37	0.76	9.29	12.71
$n = 200$	feasible	3.82	3.35	56.14	19.17
$m = 8000$	Z-inf.	0.03	0.52	4.03	14.86
$d = 10\%$	Q-inf.	0.01	0.09	0.79	4.84
all (29, 11, 20)		1.86	1.74	28.14	13.60
$n = 200$	feasible	17.76	11.01	>120.0	20.75
$m = 16000$	Z-inf.	0.04	0.65	5.59	18.13
$d = 20\%$	Q-inf.	0.01	0.10	0.84	5.19
all (28, 12, 20)		8.30	5.30	>57.4	15.04
$n = 800$	feasible	3.78	13.57	>120.0	>120.0
$m = 12800$	Z-inf.	0.12	6.80	63.98	>120.0
$d = 1\%$	Q-inf.	0.03	1.68	14.17	>120.0
all (27, 13, 20)		1.74	8.14	>72.59	>120.0

of HaSt are similar to the satisfaction case, and Impl needs about three times longer. While HaSt improves the more dense the system, Impl is the clear winner on sparse systems.

While Utvpi constraints are used in a number of practical applications, they are usually deeply embedded inside other systems, such as theorem provers, or program

Table 3.3: Average runtime in seconds of the implication algorithms

Examples		Impl	HaSt
$n = 100$	$p = 50$	0.31	1.28
$m = 1000$	$p = 100$	0.32	1.29
$d = 5\%$	$p = 200$	0.34	1.31
$n = 100$	$p = 50$	0.53	1.49
$m = 2000$	$p = 100$	0.54	1.50
$d = 10\%$	$p = 200$	0.56	1.52
$n = 100$	$p = 50$	1.10	1.66
$m = 4000$	$p = 100$	1.12	1.68
$d = 20\%$	$p = 200$	1.14	1.70
$n = 200$	$p = 100$	2.11	12.52
$m = 4000$	$p = 200$	2.16	12.56
$d = 5\%$	$p = 400$	2.26	12.66
$n = 200$	$p = 100$	3.84	12.19
$m = 8000$	$p = 200$	3.89	12.25
$d = 10\%$	$p = 400$	4.00	12.35
$n = 200$	$p = 100$	10.24	13.59
$m = 16000$	$p = 200$	10.31	13.66
$d = 20\%$	$p = 400$	10.46	13.82
$n = 800$	$p = 400$	34.74	>200.0
$m = 12800$	$p = 800$	35.50	>200.0
$d = 1\%$	$p = 1600$	37.08	>200.0

analysers, so there are no suites of stand-alone Utvpi problems we are aware of. Our experience of the kinds of Utvpi problems that arise from these applications are that they are very sparse. This indicates that our algorithms should be advantageous for real applications.

3.7 Non-Incremental Implication Checking and Generation

The incremental algorithm of the previous sections can be extended to create non-incremental implication algorithms which either check all constraints in a set P for implication or compute all (tightly) implied constraints. These algorithms are of particular importance for the use of Utvpi constraints in abstract interpretation (Miné 2006) since they allow checking of implication between two sets of Utvpi constraints, and building a canonical form of a set of Utvpi constraints.

Algorithm 3.5: Non-incremental satisfaction and implication for UTVPI constraints.

Input: $G_\phi = (V, E)$ a *constraint graph* representing set of UTVPI constraints ϕ and P a set of UTVPI constraints not implied by ϕ.

Output: Either $P' \subseteq P$ a set of UTVPI constraint implied by ϕ, or Unsat, if ϕ is unsatisfiable.

```
1 if LaMu(φ) = Unsat then return Unsat;
2 P' := ∅; let π be the generated valid potential function of G_φ;
3 for all u ∈ V do
4 |   compute δ_u^→ by using rc(G_φ) via π;
5 |   for all c ≡ u + by ≤ d ∈ P do
6 |   |   if y ≠ 0 ∧ δ_u^→(y^{-b}) ≤ d or y = 0 ∧ ⌊δ_u^→(−u)/2⌋ ≤ d then
  |   |   P' := P' ∪ {c}
7 return P'
```

The non-incremental implication algorithm which checks constraints P for implication by ϕ is shown in Alg. 3.5. At first it checks the satisfiability of ϕ by using LaMu. If ϕ is unsatisfiable then the algorithm terminates with Unsat. Then the shortest path of all pairs of nodes is computed using Johnson's algorithm (see Johnson 1977) which runs Dijkstra's algorithm from every node u in the reduced cost graph $rc(G_\phi)$ via the valid potential function π.

If the shortest path between u and v is d' then all constraints $u - v \leq d$ are implied with $d' \leq d$. To ensure that we check each constraint $ax + by \leq d \in P$ at most one time, we assume a map from $u = ax$ to all constraints of the form $ax + by \leq d \in P$.

The overall complexity is $\mathcal{O}(n^2 \log n + nm + |P|)$ time and $\mathcal{O}(n + m + |P|)$ space which is determined by Johnson's algorithm $\mathcal{O}(n^2 \log n + nm)$ time and the size of the implication set P.

To generate all tightly implied constraints of satisfiable system ϕ, that is for $\phi \models ax + by \leq d$ but $\phi \not\models ax + by \leq d'$ for $d' < d$, we use a variation of the same algorithm. Instead of checking the constraints in P for implication we use $\delta_u^\rightarrow$ to create new constraints. For each $u \in V$ and each v where $\delta_u^\rightarrow(v) < +\infty$ we create the constraint $u - v \leq \delta_u^\rightarrow(v)$ if $v \notin \{u, -u\}$, and the constraint $u \leq \lfloor \delta_u^\rightarrow(v)/2 \rfloor$ when $v = -u$.

Finally we generate $u + v \leq d_u + d_v$ for each $u \leq d_u$ and $v \leq d_v$ previously created where $v \notin \{u, -u\}$, and remove any quasi-syntactic redundant constraints, that is $ax + by \leq d$ where $ax + by \leq d'$ and $d' < d$, from the generated set.

This algorithm needs $\mathcal{O}(n^2 \log n + nm)$ time and $\mathcal{O}(n + m + |P|)$ space, since the number of implied constraints $|P|$ is bounded in $\mathcal{O}(n^2)$.

3.8 Generation of Minimal Unsatisfiable Subsets and Minimal Implicants

Given ϕ an unsatisfiable set of UTVPI constraints, then a minimal unsatisfiable subset of ϕ is a set $M \subseteq \phi$ such that M is unsatisfiable and each $M' \subset M$ is satisfiable. Suppose that $\phi \models c$, a minimal implicant M of c is a set $M \subseteq \phi$ such that $M \models c$ and for each $M' \subset M$, $M' \not\models c$. These are highly related since $M \models c$ if and only if $M \wedge \neg c$ is unsatisfiable. Minimal unsatisfiable subsets (minimal implicants) are useful if we are using a UTVPI solver as a theory solver in a lazy online SMT solver (Nieuwenhuis *et al.* 2005) which requires an explanation of unsatisfiability (and implication) to encode in Booleans the knowledge of the UTVPI solver.

In the remainder of this section we explain the generation of a minimal subset $M \subseteq \phi$ in the case of $\mathbb{Q}$-unsatisfiability of ϕ. A minimal subset generator for $\mathbb{Z}$-unsatisfiability and implication $\phi \models c$ can be adapted easily from the $\mathbb{Q}$-unsatisfiability case.

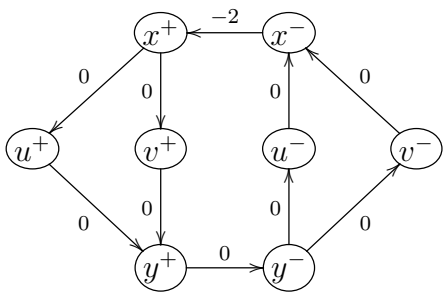
The underlying idea is to use the algorithm INCCONDIFF to recover a simple negative cycle as in Cotton and Maler (2006) and construct a minimal unsatisfiable set with respect to this cycle. The algorithm recovers a negative cycle by keeping track of the last edge (u, v, d) for every node v which refines $\gamma(v)$ in INCCONDIFF. Hence, each node in a negative cycle has one associated edge which will form a simple negative cycle. This cycle is extracted by backtracing from x to y where $(x, y, .)$ is the last added edge to the constraint graph.

The corresponding constraint set of the cycle defines a minimal unsatisfiable set in the difference constraint context, but not in all cases for UTVPI constraints, since a UTVPI constraint with two variables is represented by two edges in the constraint graph.

Example 3.8. Consider the satisfiable constraint set $\phi = \{x - u \leq 0, u - y \leq 0, x - v \leq 0, -v - y \leq 0, y \leq 0\}$ and the constraint $c \equiv -x \leq -1$. Here $\phi' = \phi \cup \{c\}$ is unsatisfiable in $\mathbb{Q}$.

The left-hand side in Fig. 3.2 shows the constraint graph $G_{\phi'}$ and the right-hand side a table with a possible sequence of INCCONDIFF steps under the assumption that the algorithm was called with G_ϕ, π, and $(x^-, x^+, -2)$ where $\pi(v) = 0$ for all nodes v. The table shows: the number of the step, the dequeued node, its tracked edge, and the caused refinements of γ.

The constraint set of the tracked negative cycle (the outer cycle in the graph $G_{\phi'}$) is ϕ' which is not minimal, since $\phi' \backslash \{x - u \leq 0, u - y \leq 0\}$ and $\phi' \backslash \{x - v \leq 0, -v - y \leq 0\}$ are the only minimal unsatisfiable sets.



No.	node	tracked edge	refines
1	x^+	$(x^-, x^+, -2)$	$\gamma(u^+) = -2$
			$\gamma(v^+) = -2$
2	u^+	$(x^+, u^+, 0)$	$\gamma(y^+) = -2$
3	v^+	$(x^+, v^+, 0)$	
4	y^+	$(u^+, y^+, 0)$	$\gamma(y^-) = -2$
5	y^-	$(y^+, y^-, 0)$	$\gamma(u^-) = -2$,
			$\gamma(v^-) = -2$
6	v^-	$(y^-, v^-, 0)$	$\gamma(x^-) = -2$
-	x^-	$(v^-, x^-, 0)$	

Figure 3.2: The outer cycle is one possible tracked cycle whose corresponding constraint set is not minimal. The steps of IncConDiff are shown on the right-hand side.

The reason for the tracked non-minimal set depends on the two equal-weight paths and theirs counterpaths from x^+ to y^+ and y^- to x^- respectively. Each path can refine the γ-value y^+ and x^- respectively, whose tracked edge decides which path we backtrace during recovery of the cycle. In our example we backtrace the path via v^- from x^- to y^-, but not its counterpath from y^+ to x^+, since the backtracked edge $(u^+, y^+, 0)$ of y^+ is part of the other path which leads to a non-minimal set. □

To generate a minimal unsatisfiable subset M of non-minimal unsatisfiable sets N we could apply a general algorithm for minimal unsatisfiability, for example that described in Junker (2004) which needs $\mathcal{O}(|N|)$ steps. In each step the algorithm checks the satisfiability of a subset of N, so the overall run-time is $\mathcal{O}(|N| \cdot \tau)$ where τ is the run-time complexity of one satisfiability check. In our case τ is $\mathcal{O}(nm)$ where n is the number of variables in N and $m = |N|$.

But we can do far better. Figure 3.3 shows the only two possible patterns of constraint graphs G_N which can occur if the corresponding Utvpi constraint set N of a simple negative cycle is not minimal. A wiggly line represents a path between two nodes and a path labeled $\bar{P}$ is the counterpath of P.

Lemma 3.9. *Let C be a simple negative cycle in G_ϕ and N be its corresponding constraint set. If N is a non-minimal unsatisfiable constraint set then its constraint graph G_N fits into one of the patterns appearing in Fig. 3.3.*

Proof. Let N be the corresponding constraint set of a simple negative cycle $C = (x_1, x_2, \ldots, x_n)$ in G_ϕ, which is a non-minimal unsatisfiable constraint set. Since $0 \leq d \notin N$ with $d < 0$ and C is a simple cycle there exists a minimal unsatisfiable constraint set $M \subset N$ with $|M| > 1$. Due to Lem. 3.2 G_M contains a simple negative

Figure 3.3: Two possible patterns of constraint graph of a non-minimal unsatisfiable constraint set arising from a simple negative-weight cycle. In Pattern A $PQRS$ is a negative cycle but $P\bar{R}$ may represent a negative cycle derived from a strict subset of the constraints. In Pattern B $PQRS$ is a negative cycle but either $PQR\bar{Q}$ or $P\bar{S}RS$ may be negative cycles derived from a strict subset of the constraints.

cycle C_M for which the following holds:

$$C_M \cap C \neq \emptyset \text{ and } C_M \setminus C \neq \emptyset \ .$$

Let $\bar{S} = (-u = -s_k \to -s_{k-1} \to \cdots \to -s_1 = -v)$ be a path in $C_M \setminus C$ such that nodes $\{-s_k, -s_1\}$ appear in C and nodes $-s_i, 1 < i < k$ do not appear in C. Moreover, let $C_{\bar{S}}$ be its corresponding constraint set. As $C_{\bar{S}} \subset C_M \subset C$ the counterpath $S = (v = s_1 \to s_2 \to \cdots \to s_k = u)$ must be a part of C.

W.l.o.g. let $x_i = s_{i-n+k}$ for all $n - k < i \leq n$, $x_i = -s_k$, and $x_j = -s_1$. Hence, $i \neq j$ and $i, j \leq n - k$.

Pattern A ($i > j$): Therefore, C equals $PQRS$, where $P = x_n \rightsquigarrow x_j$, $Q = x_j \rightsquigarrow x_i$, and $R = x_i \rightsquigarrow x_{n-k+1}$.

Pattern B ($i < j$): For pattern B the paths on $C = PQRS$ are $P = x_n \rightsquigarrow x_i$, $Q = x_i \rightsquigarrow x_j$, and $R = x_j \rightsquigarrow x_{n-k+1}$.

Thus, G_C also includes their counterpath in both cases. Furthermore, because of the selection of $\bar{S}$ the paths Q, $\bar{Q}$, S, and $\bar{S}$ do not share any interior nodes. Note that this is not necessarily the case for the pairs of paths P, $\bar{R}$, and $\bar{P}$, R. This means that all paths from a node in Q, S to a node in P, or R must pass at least u, $-u$, v, or $-v$. □

Our approach to determining minimal unsatisfiable subsets extends Cotton and Maler (2006) by preventing the construction of negative cycles including pattern A or B. It does do by slightly changing the order for dequeuing the nodes and extending the backtracing of a negative cycle, so that the generated unsatisfiable constraint

set is minimal. First we introduce a new order $\prec$ on the pair $(\gamma, \#_s)$ over γ and the minimal number of edges of a shortest path from s to the nodes:

$$(\gamma(u), \#_s(u)) \prec (\gamma(v), \#_s(v)) \quad \Longleftrightarrow \quad \gamma(u) < \gamma(v) \text{ or } \gamma(u) = \gamma(v) \wedge \#_s(u) < \#_s(v)$$

where $\#_s(x) = \min\{|P| \mid P = s \rightsquigarrow x : w(P) = wSP(s, x)\}$.

Compared to the order used by IncConDiff this order changes the sequence of dequeuing of nodes—therefore also for the tracked edges, as well—but only for two nodes with the same γ-values. The behaviour of dequeuing on those nodes is like a breadth-first search. Thus, IncConDiff with $\prec$ will find a simple negative cycle with the shortest length and the minimal number of edges, so that the constraint graph of its corresponding constraints cannot match with pattern A.

Example 3.9. Consider the pattern A of Fig. 3.3 and suppose that all paths only include edges with a weight of 0 except for P and $\bar{P}$ which only contain the edge $(u, -v, -2)$ and $(v, -u, -2)$ respectively. Moreover, let us assume that no path shares any of its interior nodes with any other path.

The graph with the paths Q, R, S, and their counterpaths has no negative cycle, but with the path P and $\bar{P}$ it does. Then a minimal unsatisfiable set can only result from the cycles $P\bar{R}$ or $\bar{P}R$. During a run of IncConDiff each γ-value can be refined exactly once to -2. The new order $\prec$ avoids a refinement of $\gamma(u)$ through the last edge in the path S, which would lead to a non-minimal set, if IncConDiff starts at $-v$. In this example $\gamma(u)$ will always be refined by the last edge in $\bar{R}$, since the number of edges of the simple shortest path from $-v$ via $\bar{R}$ to u is smaller by $|QS|$ than the other simple path QRS. □

To avoid a recovery of a cycle whose constraint graph fits in pattern B we change the tracked edge of some nodes during backtracing of a cycle as follows: If we backtrace over the edge (u, v, d) then we set the tracked edge of $-u$ to the counteredge $(-v, -u, d)$ if and only if $0 = \pi(-v) + d - \pi(-u)$ and $\#_s(-u) = \#_s(-v) + 1$. That is, if the path from s via $-v$ to $-u$ is a simple shortest path from s to $-u$ and the number of edges of both paths are the same. (Only in this situation can IncConDiff make a "bad" decision concerning pattern B during the $\mathbb{Q}$-satisfiability check.)

Example 3.10. Let us consider the Ex. 3.8. The steps shown in the right of Fig. 3.2 can still be performed by using the new order $\prec$, because the competing paths have the same number of edges.

During the recovery of the cycle—beginning at the node x^-—we switch the tracked edge $(u^+, y^+, 0)$ of y^+ to $(v^+, y^+, 0)$, since we backtrace over the edge $(y^-, v^-, 0)$

Algorithm 3.6: ModIncConDiff – Modified version of IncConDiff returning a minimal unsatisfiable set in the case of $\mathbb{Q}$-unsatisfiability.

Input: $G_\phi = (V, E)$ a graph, π a valid potential function for G_ϕ, (u, v, d) a new edge to add to G_ϕ.

Output: Minimal unsatisfiable set M if $\phi \cup \{u - v \leq d\}$ is unsatisfiable, or $G_{\phi \cup \{u-v \leq d\}}$ and a valid potential function π' for $G_{\phi \cup \{u-v \leq d\}}$.

```
1  γ(v) := π(u) + d − π(v); #_v(v) := 0;
2  γ(w) := 0 and #_v(w) := ∞ for all w ≠ v;
3  while min(γ) < 0 ∧ γ(u) = 0 do
4      s := argmin(γ, #_v) wrt. the order ≺ ;
5      π'(s) := π(s) + γ(s) ;
6      γ(s) := 0 ;
7      for all s --d'--> t ∈ G do
8          if π'(t) = π(t) and γ(t) > π'(s) + d' − π'(t) then
9              Δ := π'(s) + d' − π'(t);
10             if γ(t) > Δ or (γ(t) = Δ and #_v(t) > #_v(s) + 1) then
11                 γ(t) = Δ;
12                 #_v(t) = #_v(s) + 1;
13                 tracked_edge(t) := (s, t, d');
14 if γ(u) < 0 then
15     s := u; M := {u − v ≤ d};
16     while s ≠ v do
17         (w, t, d') := tracked_edge(s);          /* t is the current s */
18         if π(−t) + d' − π(−w) = 0 and #_v(−w) = #_v(−t) + 1 ≠ ∞ then
19             tracked_edge(−w) := (−t, −w, d');
20         M := M ∪ {w − t ≤ d'};
21         s := w;
22     return M
23 return ((V, E ∪ {(u, v, d)}), π')
```

before reaching y^+. Thus, the recovered cycle contains the path via v^- and its counterpath instead of its competing path via u^+, so that the corresponding constraint set $\{-x \leq -1, x - v \leq 0, -v - y \leq 0, y \leq 0\}$ is a minimal unsatisfiable set. □

The modified version of IncConDiff with $\prec$ and backtracing is presented in ModIncConDiff. In the $\mathbb{Q}$-unsatisfiability case the algorithm returns a minimal unsatisfiable set. Otherwise, it is the same as IncConDiff.

Theorem 3.8. *Let ϕ be a satisfiable set of* Utvpi *constraints and c be a* Utvpi *constraint. If $\phi \cup \{c\}$ is $\mathbb{Q}$-satisfiable then each run of* ModIncConDiff *which adds one edge of $E(c)$ more to G_ϕ is successful and the last run establishes a valid*

potential function on $G_{\phi\cup\{c\}}$; Otherwise one run of ModIncConDiff *terminates with a minimal unsatisfiable set M of $\phi \cup \{c\}$.*

Proof (sketch): The correctness of the algorithm—whether it terminates with $G_{\phi\cup\{c\}}$ and a valid potential function π on that graph or it terminates with a minimal unsatisfiable set M—follows straightforwardly from the correctness of IncConDiff, since $\prec$ only changes the order of dequeuing the nodes where the "old" order chooses "randomly" the next node to dequeue.

To show that the algorithm generates a minimal unsatisfiable set M if $\phi \cup \{c\}$ is unsatisfiable leads to an extensive case study. For this reason we omit the details of this proof.

We need to show that because of the order $\prec$ the corresponding constraint graph of the recovered negative cycle by ModIncConDiff cannot match with the pattern A. Then we show that this corresponding graph also cannot fit in the pattern B due to the switching of the tracked edge during the backtracing of the cycle.

Due to Lem. 3.9 the tracked cycle must lead to a minimal unsatisfiable set. □

The modified algorithm ModIncConDiff can also be used to generate a minimal unsatisfiable set in the $\mathbb{Z}$-unsatisfiable case, and a minimal implicant. For both cases we run the algorithm on ϕ with the edges in $E(c')$ of the constraint $c' \equiv ax + by \leq d-2$, if $c \equiv ax+by \leq d$ causes the $\mathbb{Z}$-unsatisfiability of $\phi\cup\{c\}$ or $c \equiv -ax-by \leq -d$ is implied by ϕ, since $\phi \cup \{c'\}$ is $\mathbb{Q}$-unsatisfiable. The complexity of answering all these questions using ModIncConDiff is $\mathcal{O}(n \log n+m)$ time and $\mathcal{O}(n+m)$ space where $m = |N|$ and n is the number of variables occurring in N.

3.9 Final Remarks

We have presented new incremental algorithms for Utvpi constraint satisfaction and implication checking which improve upon the previous asymptotic complexity, and perform better in practice for sparse constraint systems.

We have adapted the algorithms herein to provide non-incremental implication checking in $\mathcal{O}(n^2 \log n + nm + p)$ time and $\mathcal{O}(n + m + p)$ space, and generate all implied constraints in $\mathcal{O}(n^2 \log n + nm)$ time and $\mathcal{O}(n + m + p)$ space, where p is the number of implied constraints generated.

We also extended the algorithms to return a minimal unsatisfiable subset when unsatisfiability is detected, and a minimal implicant of an implied constraint.

4

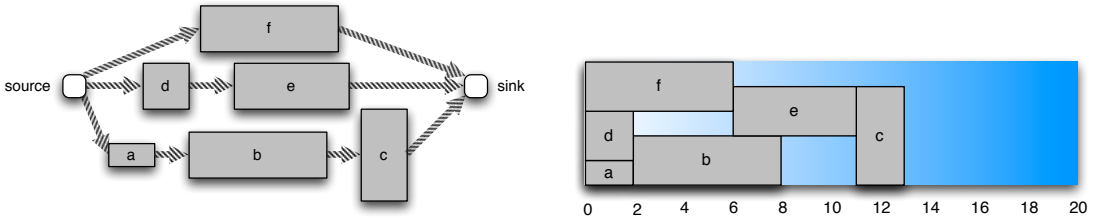
Explaining the Propagation of the Cumulative Constraint

The cumulative constraint models the usage of scarce cumulative resources. These resources are part of many real-world scheduling problems such as constructing a bridge or building, employee scheduling, ship loading, time tabling, processor or production scheduling etc. (see e.g. Neumann and Schwindt 1997, Kolisch 2001, Demeulemeester and Herroelen 2002, Artigues *et al.* 2008). Moreover, each dimension in multi-dimensional cutting and packing problem can be seen as an individual cumulative resource. Thus, cumulative resources can describe not only machines that are able to run multiple tasks in parallel but also entities such as: electricity, water, consumables or even human skills.

In this chapter we investigate how to explain the propagation of the cumulative constraint, so that the explanations lead to a pruning of the search space as much as possible. These explanations are developed for cumulative propagators in an LCG solver (described in Sec. 2.4 on page 30). Before this we study explanations that are created when using decompositions of the global constraint `cumulative`, and use the knowledge gained for the creation of explanations for `cumulative`. Experimental results are presented in Chap. 5 and 6.

4.1 Introduction

The cumulative constraint models the relationship between a scarce *cumulative resource* and *activities* requiring some part of the resource capacity for their execution. The resource cannot be overloaded at any point of time, *i.e.*, the accumulated resource requirements of activities must be less than or equal to the available resource



(a) Activities with precedence relations.

(b) A possible schedule.

Figure 4.1: A small cumulative resource problem, with 6 activities to place in the 5x20 box, with activity `a` before `b` before `c`, and activity `d` before `e`.

capacity in each time period. In this work, we restrict ourself to *renewable* resources, *i.e.*, resources with a constant resource capacity over time, and *non-preemptive* activities, *i.e.*, their execution cannot be interrupted. A change in the resource capacity can be modelled by fictitious activities that claim the non-available capacity in a time interval.

Example 4.1. Consider a simple resource scheduling problem. There are six activities `a`, `b`, `c`, `d`, `e`, and `f` to be scheduled to end before time 20. The activities have respective durations 2, 6, 2, 2, 5, and 6, and respective activities requiring 1, 2, 4, 2, 2, and 2 units of resource, with a resource capacity of 5. Assume further that there are precedence constraints: activity `a` must complete before activities `b` begins, written $\texttt{a} \ll \texttt{b}$, and similarly $\texttt{b} \ll \texttt{c}$, $\texttt{d} \ll \texttt{e}$. Figure 4.1a shows the six activities and precedences, while 4.1b shows a possible schedule, where the last activity ends at time period 13. □

In 1993 Aggoun and Beldiceanu (1993) introduced `cumulative` in order to efficiently solve complex scheduling problems in a constraint programming framework, especially for FD solvers. The cumulative constraint cannot compete with specific Operations Research methods for restricted forms of scheduling, but since it is applicable whatever the side constraints are it is very valuable. Since that time a great deal of research has investigated new stronger and faster propagation techniques (see e.g. Caseau and Laburthe 1996, Carlier and Pinson 2004, Nuijten 1994, Baptiste and Le Pape 2000, Schutt and Wolf 2010, Vilím 2009), but still most of these techniques only pay off in limited cases or are not scalable.

Demeulemeester and Herroelen (1992, 1997) implicitly show the importance of nogoods to fathom the huge search space of the basic resource-constrained project scheduling (RCPSP) problem that involves several cumulative resources and precedence relations between activities. Their method is the best-known complete method so far and heavily relies on dominance rules and cut sets—a kind of problem specific nogoods—but unfortunately, the number of cut sets grows exponentially in the

number of activities, so that this method is considered to be efficient only for small problems.

This observation indicates that an explaining version of `cumulative` may significantly improve current scheduling methods. In comparison to the specific nogoods of Demeulemeester and Herroelen (1992, 1997) we are interested in general nogoods, since they are domain-independent. As discussed in Sec. 2.3, SAT solvers learn general nogoods from each conflict, but a static translation of a problem to a SAT problem may be prohibitive because of the implicit limits on the number of variables and constraints that they can handle efficiently. In practice, this is the case for problems involving cumulative resources. However, LCG solvers and lazy online SMT solvers (see Sec. 2.4 and 2.5) allow the usage of the SAT solver facilities by lazily transforming the problem to a SAT problem. Ohrimenko *et al.* (2009) use an LCG solver to solve open shop scheduling problems—which is a special case of RCPSP—by decomposition, their approach outperforms the best available constraint solvers. Consequently, explanation for `cumulative` in an LCG solver seems to be very promising.

Global constraints can almost always be *decomposed* into simpler constraints by introducing new intermediate variables to encode the meaning of the global constraint. Since the introduction of `cumulative` (Aggoun and Beldiceanu 1993), little attention has been paid to decompositions of `cumulative` because decomposition cannot compete with the global propagator because of the overheads of intermediate variables, and the lack of a global view of the problem. But once we consider explanation we have to revisit this. Decomposition of global constraints means that explanation of behaviour is more fine grained and hence more reusable. Also it avoids the need for complex explanation algorithms to be developed for the global constraint.

In this chapter we investigate how to build explanations for `cumulative` in an LCG solver, so that these explanations lead to stronger nogoods deduced during the conflict learning in the SAT solver. Firstly, the cumulative resource scheduling is introduced and related work of explanation for `cumulative` is discussed. Secondly, we examine two decompositions of `cumulative`. Then, we show that building `cumulative` with specialised explanation capabilities can further improve upon the explaining decompositions. Finally, the explanations of `cumulative` are extended to take flexible resource capacities, flexible resource usages of activities, and flexible processing times of activities into account.

4.2 Cumulative Resource Scheduling

First, we define activities and the cumulative resource scheduling problem, and then discuss algorithms for the cumulative propagator.

Definition 4.1 (Activity)**.** An *activity* (also called *task*) i is specified by its *start time* s_i, its *processing time* (*duration*) p_i, its *resource usage* (*capacity usage* or *capacity/resource requirement*) r_i, and its *energy* (*area* or *workload*) $energy_i = p_i \times r_i$.

In general, the start time of an activity is variable. Depending on the problem, the processing time and resource usage can be variable too while the energy is fixed. In rare circumstances (e.g. steel melting or carpet cutting) the processing time and energy can be variable, but not the resource usage.

Definition 4.2 (Cumulative Resource (Scheduling) Problem)**.** A *cumulative resource (scheduling) problem* (Crp) is a Csp that is characterised by a cumulative resource with the (constant) *resource capacity* R, a set of (non-preemptive) activities $\mathcal{V}$, and a domain on the activity variables. The goal is to find a solution that assigns values from the domain to the variables for each activity, so that the following constraints are satisfied:

$$\forall i \in \mathcal{V}: \quad energy_i = p_i \times r_i \tag{4.1}$$

$$\forall \tau \in [0..T): \quad \sum_{i \in \mathcal{V}: s_i \leq \tau < s_i + p_i} r_i \leq R \ , \tag{4.2}$$

where T is the *planning horizon* and τ is a *time period* or, short, a *period*. A solution is also called *schedule*.

If just the start times are variable then the goal simplifies to finding an assignment for the start time variables that satisfies (4.2), and (4.1) is an equation between integers.

In the remainder of this work, models or part of them are stated in the Zinc modelling language (Marriott *et al.* 2008). The constraint `cumulative` has the Zinc type

```
predicate cumulative(array[int] of var int: s,
  array[int] of var int: p, array[int] of var int: r, var int: R);
```

where `s[i]` is the start time of the activity i, `p[i]` its processing time, `r[i]` its resource usage, and `R` the resource capacity.

Example 4.2. Consider the cumulative resource problem defined in Ex. 4.1. This can be modelled by

$$\texttt{cumulative}([s_\texttt{a}, s_\texttt{b}, s_\texttt{c}, s_\texttt{d}, s_\texttt{e}, s_\texttt{f}], [2, 6, 2, 2, 5, 6], [1, 2, 4, 2, 2, 2], 5)$$

with precedence constraints $\texttt{a} \ll \texttt{b}$, $\texttt{b} \ll \texttt{c}$, $\texttt{d} \ll \texttt{e}$, modelled by $s_\texttt{a} + 2 \leq s_\texttt{b}$, $s_\texttt{b}+6 \leq s_\texttt{c}$, and $s_\texttt{d}+6 \leq s_\texttt{e}$. The propagators for the precedence relations determines a domain D where $D(s_\texttt{a}) = [\,0\,..\,8\,]$, $D(s_\texttt{b}) = [\,2\,..\,10\,]$, $D(s_\texttt{c}) = [\,8\,..\,18\,]$, $D(s_\texttt{d}) = [\,0\,..\,13\,]$, $D(s_\texttt{e}) = [\,2\,..\,15\,]$, $D(s_\texttt{f}) = [\,0\,..\,14\,]$. The cumulative propagator does not determine any new information. If we add the constraints $s_\texttt{c} \leq 9$, $s_\texttt{e} \leq 4$, then precedence determines the domains $D(s_\texttt{a}) = [\,0\,..\,1\,]$, $D(s_\texttt{b}) = [\,2\,..\,3\,]$, $D(s_\texttt{c}) = [\,8\,..\,9\,]$, $D(s_\texttt{d}) = [\,0\,..\,2\,]$, $D(s_\texttt{e}) = [\,2\,..\,4\,]$. The cumulative propagator may be able to determine that activity `f` cannot start before time 10. □

A special case of CRP is disjunctive resource scheduling where the resource capacity R equals 1. Although all discussions about CRP hold for this special case in this work we concentrate on pure CRP, *i.e.*, $R > 1$, since more sophisticated techniques exist to tackle scheduling problems with disjunctive resources. In disjunctive resource scheduling an activity has to precede or follow another activity, since activities cannot be executed concurrently. Relying on this fact, the explanation developed later in this chapter might not be the best one for solving such problems. Some other solution methods can exploit the binary relationship between activities.

To decide whether a CRP problem is feasible is NP-complete, since already its specialised version where all parameters of activities are fixed except the start times is NP-complete (Baptiste *et al.* 1999). Thus, there does not exist an efficient algorithm that can solve all CRP instances in polynomial time, unless $P = NP$. Consequently, no efficient cumulative propagator can even achieve bounds($\mathbb{R}$) consistency (cf. Def. 2.12 on page 13) (unless $P = NP$).

Nonetheless, different propagation algorithms have been developed which achieve a weaker consistency level than bounds($\mathbb{R}$) consistency. They differ in their propagation strength, their runtime complexity, and their applicability. Baptiste *et al.* (2001) exhaustively relates them to each other regarding their propagation strength. The propagation algorithms mainly reason about the compulsory parts of activities (Lahrichi 1982) or about the energies of activities (Erschler and Lopez 1990). These algorithms are referred as *consistency check* or *filtering* algorithms depending on their purpose.

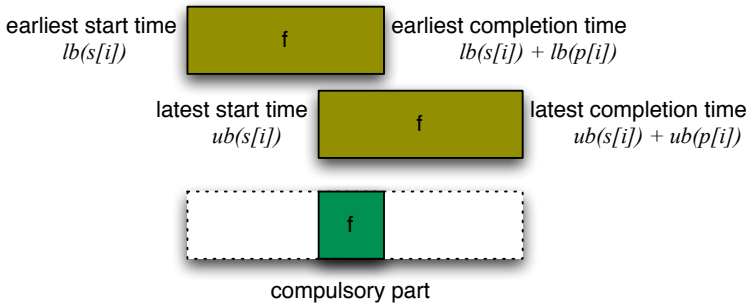


Figure 4.2: The compulsory part of an activity (on the bottom) deduced from the earliest start time (on the top) and the latest completion time (on the bottom).

4.2.1 Reasoning about the Compulsory Parts

If $ub(s_i) < lb(s_i) + lb(p[i])$ then the *compulsory part* of an activity i is the time interval $[ub(s_i), lb(s_i) + lb(p[i]))$ in which the activity must be executed due to its time window. The compulsory part (see Fig. 4.2) can be seen as an intersection between the schedule times if the activity is started at its earliest and latest time considering the lowest and highest processing time of the activity, respectively.

The consistency check based on these parts is called *time-table* (or *time-tabling*). It records the compulsory parts of all activities over the planning horizon. This record is called the *resource profile* (or *histogram*). This check can be performed in $\mathcal{O}(n \log n)$ runtime and $\mathcal{O}(n)$ space where n is the number of activities.

The *time-table filtering* algorithm uses this resource profile to infer new bounds on the start and end time variables. For each activity it tests if the activity can be started at its earliest start time without causing a resource overload. If not then the lower bound of the start time variable is increased appropriately. The case for the upper bound of the start time variable is symmetric. The time complexity of these algorithms is $\mathcal{O}(n \log n + np)$ where p is the number of height changes in the profile.

Due to their low complexity and their usage of simple data structures both algorithms appear to be the most scalable algorithms for the cumulative propagator so far. Therefore they are usually used as default propagators. A drawback is that the search must lead to creation of compulsory parts before an inference can be made. Thus, the main propagation may start late in the search.

4.2.2 Reasoning about the Energies

In comparison to the time-table algorithms, reasoning about energies of the activities can deduce new bounds in an early stage of a search.

The overload check (Wolf and Schrader 2006) is a consistency check algorithm. It checks for a resource overload in specific time intervals that are defined by the activities. In each time interval it sums up the needed energies of the activities that must be completely scheduled in this interval and checks if the sum exceeds the available energy. It runs in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

Nuijten (1994) generalised the filtering algorithms of (extended) edge-finding, and not-first/not-last for the disjunctive propagator to the cumulative propagator. All three algorithms detect the relationship between one activity j and a subset of activities Ω excluding j by considering the activity's energy. (Extended) edge-finding infers if the activity j must strictly end after (start before) all activities in Ω, not-first infers if the activity j must start after the end of at least one activity in Ω and not-last infers if the activity j must end before the start of at least one activity in Ω.

The edge-finding algorithm was corrected by Mercier and Van Hentenryck (2008) and its runtime complexity reduced to $\mathcal{O}(kn \log n)$ (Vilím 2009) where k is the number of distinct resource usages of activities. The not-first/not-last algorithm was corrected by Schutt *et al.* (2006) and its runtime complexity lowered to $\mathcal{O}(n^2 \log n)$ (see Schutt and Wolf 2010).

Baptiste and Le Pape (2000) present the left-shift/right-shift filtering algorithm, also called energetic reasoning (Baptiste *et al.* 2001), which subsumes (extended) edge-finding, but not not-first/not-last. It considers the required energies of activities in specific time intervals. This required energy is the minimal energy that overlaps with the time interval when the activity starts at its earliest start time (left shift) or ends at its latest completion time (right shift). It is more complex and stronger than the other filtering algorithms, but it needs to consider many time intervals to demonstrate its full power. Since it runs in $\mathcal{O}(n^3)$ time, it is rarely used.

Caseau and Laburthe (1996) generalised the use of activity intervals from the disjunctive constraint to the cumulative constraint. An activity interval is characterised by two activities i and j and contains all activities whose earliest start time and latest end time are included in the time interval $[\, start \,..\, end \,]$ where *start* is the earliest start time of i and *end* the latest end time of j, *i.e.*, $ub(s[j]) + ub(p[j])$. The number of activity intervals is thus $\mathcal{O}(n^2)$. The activity intervals are incrementally maintained and used to check consistency, propagate the lower (upper) bound of the start (end) variable by rules which cover (extended) edge-finding, detect precedences between activities and eliminate duration-usage pairs if the energy for an activity is given. Additionally, the compulsory parts are incrementally tracked for the consistency check and the time-table filtering.

These filtering algorithms are more likely to propagate if the *slack*, *i.e.*, available

energy in some time interval, is small. Baptiste and Le Pape (2000) showed that the use of these algorithms is beneficial for highly cumulative problems, *i.e.*, problems in which many activities can be run in parallel, but not for highly disjunctive problems. Due to its lower complexity and simpler structure the edge-finding algorithm is the most used one if the problem is not highly disjunctive.

4.3 Related Work on Explanations

There is a substantial body of work on explanations in constraint satisfaction (see e.g. Dechter *et al.* 1991, Chap. 6), but there was little evidence until recently of success for explanations that combine with propagation (although see Jussien *et al.* 2000, Jussien and Lhomme 2002). The constraint programming community revisited this issue after the success of explanations in the SAT community.

Katsirelos and Bacchus (2005) generalised the nogoods from the SAT community. Their generalised nogoods are conjunctions of variable-value equalities and disequalities, *e.g.* $\{x_1 = 3, x_4 = 0, x_7 \neq 6\}$. For bound inference algorithms this representation is not suitable since one bound update of a variable must be explained by several nogoods.

The LCG approach (introduced in Sec. 2.4 on page 30) is a hybrid of SAT and FD solvers. It keeps the abstraction of the constraints and their propagators just explain their inferences to the SAT solvers. Moreover, their bounds on integer variables are represented by Boolean literals so they are more suitable for explaining a bound filtering algorithm. In this chapter we use LCG to implement a first version of an explaining global cumulative constraint consisting of the time-table consistency check and filtering algorithm. This constraint is compared to the two decompositions which also use LCG on the parts of their decomposition.

There has been a small amount of past work on explaining `cumulative`. Vilím (2005) considered the disjunctive case of `cumulative` where he presented a framework based on justification and explanation. The explanation consists of a subset of initial constraints, valid search decisions, and conflict windows for every activity. He proposed explanations for an overload check concerning an activity interval, edge-finding, not-first/not-last, and detectable precedence filtering algorithms.

In this work, we consider the general case `cumulative` and in particular show how to explain time-table and edge-finding filtering. Explaining edge-finding for general `cumulative` is more complex than for the disjunctive case, since we have to take into account the resource capacity and that activities can run in parallel. Moreover, Vilím (2005) does not consider explaining the propagation of the filtering algorithms

stepwise.

Jussien (2003) presents explanations for the time-table consistency and filtering algorithms in the context of the PaLM system (Jussien and Barichard 2000). The system explains inconsistency at time t by recording the set of activities S_t whose compulsory part overlaps t and then requiring that at least one of them takes a value different from their current domain. These explanations are analogous to the naïve explanations we describe later which examine only the current bounds, but they are much weaker since they do not use bounds literals in the explanations. The time-table filtering explanations are based on single time periods but again use the current domains of the variables to explain.

Recently, independently from our work, explanations for `cumulative` have been developed in the Scip framework (Achterberg 2009), which is a hybrid of constraint and integer programming (Berthold *et al.* 2010, Heinz and Schulz 2011). Their explanations are built "backwards", i.e., during backtracking, whereas the explanations herein are created "forwards", i.e., during the propagation.

They present explanations for the energetic reasoning and time-table consistency and filtering algorithms. Based on energetic reasoning, they also describe explanations for a restricted version of the edge-finding filtering algorithm. In all cases, the explanations consist of lower or upper bounds the activities involved. Besides the trivial explanations they determine a minimal set of activities that causes the inconsistency or a bound update for the considered algorithms. Neither Berthold *et al.* (2010), Heinz and Schulz (2011), described any form of bounds widening. Hence, it appears their explanations for the time-table and edge-finding algorithms correspond to the naïve explanation described later.

4.4 Propagating the Cumulative Constraint by Decomposition

Usually the cumulative constraint is implemented as a single propagator, since it can then take more information into account during propagation. But building a global constraint is a considerable undertaking which we can avoid if we are willing to encode the constraint using decomposition into primitive constraints.

In the remainder of this section we assume that only the start times are variable and the other parameters of an activity are fixed.

4.4.1 Time Decomposition

The time decomposition (*TimeD*) Aggoun and Beldiceanu (1993) (sometimes referred as time-indexed decomposition) arises from (4.2). For every time period t the sum of all resource requirements must be less than or equal to the resource capacity. The Zinc encoding of the decomposition is shown below where: index_set(a) returns the index set of an array a (here $[1..n]$), lb(x) (ub(x)) returns the declared lower (upper) bound[1] of a variable x, and bool2int(b) is 0 if the Boolean variable b is false, and 1 if it is true.

```
predicate cumulative(
    array[int] of var int: s, array[int] of var int: p,
    array[int] of var int: r, var int: R
) = let {
    set of int: V = index_set(s),
    set of int: times = min([lb(s[i]) | i in V]) ..
                        max([ub(s[i]) + ub(p[i]) - 1 | i in V])
} in forall(t in times) (
    sum(i in V)(
      bool2int( s[i] <= t /\ t < s[i] + p[i] ) * r[i]
    ) <= R
);
```

This decomposition implicitly introduces new Boolean variables B_{it}. Each B_{it} represents that activity i is active at time period t:

$$\forall t \in [0..T-1], \forall i \in [1..n]: \qquad B_{it} \leftrightarrow [\![s[i] \leq t]\!] \wedge \neg [\![s[i] \leq t - p[i]]\!]$$

$$\forall t \in [0..T-1]: \qquad \sum_{i \in [1..n]} r[i] \cdot B_{it} \leq R$$

Note that since we use LCG, the Boolean variables for the expressions $[\![s[i] \leq t]\!]$ and $[\![s[i] \leq t - p[i]]\!]$ already exist and for an activity i we only need to construct variables B_{it} where $lb(s[i]) \leq t < ub(s[i]) + p[i]$ for the initial domain D_{init}.

At most $n \times T$ new Boolean variables, $n \times T$ conjunction constraints, and T inequality constraints (of size n) are created. This decomposition implicitly records the resource profile over time.

To add another cumulative constraint for a different resource on the same activities we can reuse the Boolean variables, and we just need to create T new sum constraints.

The variable B_{it} records whether the activity i must use its resources at time period t. Hence B_{it} is true indicates a compulsory part of activity i. It holds in the

[1] The declared lower and upper bound are the bounds of the initial domain.

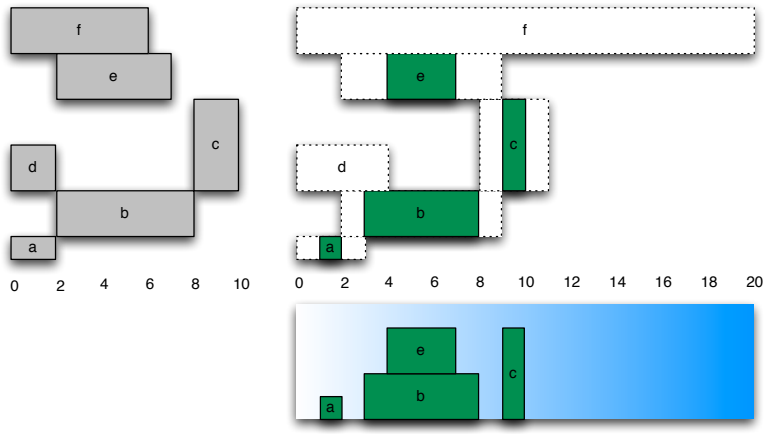


Figure 4.3: An example of propagation of the cumulative constraint.

time interval $[\, lb(s[i]) \,..\, lb(s[i]) + p[i] - 1 \,] \cap [\, ub(s[i]) \,..\, ub(s[i]) + p[i] - 1 \,]$.

Example 4.3. Consider the problem of Ex. 4.2 after the addition of $s_{\mathtt{c}} \leq 9$, $s_{\mathtt{e}} \leq 4$. The domains are $D(s_{\mathtt{a}}) = [\,0\,..\,1\,]$, $D(s_{\mathtt{b}}) = [\,2\,..\,3\,]$, $D(s_{\mathtt{c}}) = [\,8\,..\,9\,]$, $D(s_{\mathtt{d}}) = [\,0\,..\,2\,]$, $D(s_{\mathtt{e}}) = [\,2\,..\,4\,]$, $D(s_{\mathtt{f}}) = [\,0\,..\,14\,]$. Propagation on the decomposition determines that $B_{\mathtt{b}5}$ is true since $s_{\mathtt{b}} \leq 5$ and $\neg(s_{\mathtt{b}} \leq 5 - 6 = -1)$, similarly for $B_{\mathtt{e}5}$. The propagator of the inequality constraints determines that $B_{\mathtt{f}5}$ is false, and hence $\neg(s_{\mathtt{f}} \leq 5) \vee s_{\mathtt{f}} \leq -1$. Since the second half of the disjunct is false already we determine that $s_{\mathtt{f}} \geq 6$. Similarly propagation on the decomposition determines that $B_{\mathtt{c}9}$ is true, and hence $B_{\mathtt{f}9}$ is false, and hence $\neg(s_{\mathtt{f}} \leq 9) \vee s_{\mathtt{f}} \leq 3$. Since the second disjunct must be false (due to $s_{\mathtt{f}} \geq 6$) we determine that $s_{\mathtt{f}} \geq 10$.

The top of Fig. 4.3 shows each activity in a window from earliest start time to latest end time, and highlights the compulsory parts of each activity. If there is no darkened part (as for `d` and `f`) then there is no compulsory part. Propagation of the decomposition will determine $B_{\mathtt{a}1}$, $B_{\mathtt{b}3}$, $B_{\mathtt{b}4}$, $B_{\mathtt{b}5}$, $B_{\mathtt{b}6}$, $B_{\mathtt{b}7}$, $B_{\mathtt{c}9}$, $B_{\mathtt{e}4}$, $B_{\mathtt{e}5}$, and $B_{\mathtt{e}6}$ which corresponds to the compulsory parts of each activity. The resulting resource profile is shown at the bottom of Fig. 4.3. Clearly at times 5 and 9 there is insufficient resource capacity for the activity `f` with the resource usage 2. □

We can expand the model to represent holes in the domains of start times—usually CSP representation of activities does not encode holes. The literal $[\![s[i] = t]\!]$ is a Boolean representing the start time of the ith activity is t. We add the constraint

$$[\![s[i] = t]\!] \rightarrow \bigwedge_{t \leq t' < t + p[i]} B_{it'}$$

which ensures that if $B_{it'}$ becomes false then the values $\{t'-p[i]+1, t'-p[i]+2, \ldots, t'\}$ are removed from the domain of $s[i]$.

We tested this extended model on large instances of RCPSP from the PSPLib, but it neither improved the search time, the number of choice points, nor the average distance to the best known upper bound in average. This is not surprising since for RCPSP we have no propagators that can take advantage of the holes in the domain. The expanded model may be useful for cumulative scheduling problems with side constraints that can benefit from such reasoning.

Another Time Decomposition

Hooker (2005) uses a variant of *TimeD* in his linear programming formulation which avoids the introduction of 0-1 variables corresponding to the Boolean variables B_{it}. His decomposition solely relies on 0-1 variables corresponding to the variables $[\![s[i] = t]\!]$ and can be written as the following Zinc predicate.

```
predicate cumulative(
    array[int] of var int: s, array[int] of var int: p,
    array[int] of var int: r, var int: R
) = let {
    set of int: V = index_set(s),
    set of int: times = min([lb(s[i]) | i in V]) ..
                        max([ub(s[i]) + ub(p[i]) - 1 | i in V])
} in forall(t in times) (
    sum(i in V)(
      sum(ti in times where t - p[i] < ti /\ ti <= t)(
        bool2int( s[i] = ti ) * r[i]
      )
    ) <= R
);
```

We compared both decompositions on some harder instances of RCPSP with 30 activities from the PSPLib by using an LCG solver. On all these instances Hooker's variant significantly slowed down the solver (a factor of 19 for one instance). There are a few hypotheses why this variant does not perform well for LCG solvers.

First, the variant can save at most $n \times T$ Boolean variables, but it can add up to $2n \times T$ clauses, due to the fact that the variables $[\![s[i] = t]\!]$ are created on demand in an LCG solver whereas the variables $[\![s[i] \leq t]\!]$ are always built. Consequently, up to $3n \times T$ additional clauses are generated to express the relationship between those variables.

Second, the LCG propagator for the linear inequality constraint has no knowledge that the individual Boolean variables $[\![s[i] = 0]\!]$, $[\![s[i] = 1]\!]$, ..., represent values from the same domain. Thus, the LCG propagator may not create a strong explanation.

Third, one strength of LCG comes from learning about bounds on variables which is achieved by using Boolean variables $[\![s[i] \leq t]\!]$. But the variant promotes learning about individual values in the domain of a variable.

To conclude, Hooker's variant of the time decomposition is very effective in the context of mixed-integer programming. However, it is not so effective for LCG solvers.

4.4.2 Activity Decomposition

The activity decomposition (*ActiD*) is a relaxation of the time decomposition. It ensures a non-overload of resources only at the start (or end) times which is sufficient to ensure non-overload at every time for non-preemptive activities. Therefore, the number of variables and linear inequality constraints is independent of the size of the planning horizon. It was used by El-Kholy (1996) for temporal and resource reasoning in planning. The Zinc code for the decomposition at the start times is below.

```
predicate cumulative(
    array[int] of var int: s, array[int] of var int: p,
    array[int] of var int: r, var int: R
) = let {set of int: V = index_set(s)}
in forall(j in V) (
  sum(i in V where i != j)(
    bool2int( s[i] <= s[j] /\ s[j] < s[i] + p[i] ) * r[i]
  ) <= R - r[j]
);
```

The decomposition implicitly introduces new Boolean variables: $B^1_{ij} \equiv$ "activity j starts at or after activity i starts", $B^2_{ij} \equiv$ "activity j starts before activity i ends", and $B_{ij} \equiv$ "activity j starts when activity i is running".

$$\begin{aligned} \forall j \in [1..n], \forall i \in [1..n] \setminus \{j\} : \quad & B_{ij} \leftrightarrow B^1_{ij} \wedge B^2_{ij} \\ & B^1_{ij} \leftrightarrow s[i] \leq s[j] \\ & B^2_{ij} \leftrightarrow s[j] < s[i] + p[i] \\ \forall j \in [1..n] : \quad & \sum_{i \in [1..n] \setminus \{j\}} r[i] \cdot B_{ij} \leq R - r[j] \end{aligned}$$

Note not all activities i must be considered for an activity j, only those i which can overlap at the start times $s[j]$ regarding precedence constraints—if they exist, resource constraints and the initial domain D_{init}.

Since the SAT solver does not know about the relationship among the B^1_{**} and B^2_{**} the following implied constraints can be posted for all $i, j \in [1..n]$ where $i < j$ in order to improve the propagation and the learning of reusable nogoods.

$$B^1_{ij} \vee B^2_{ij} \qquad B^1_{ji} \vee B^2_{ji} \qquad B^1_{ij} \vee B^1_{ji} \qquad B^1_{ij} \rightarrow B^2_{ji} \qquad B^1_{ji} \rightarrow B^2_{ij}$$

In addition for each precedence constraint $i \ll j$ we can post $\neg B_{ij}$.

The size of this decomposition only depends on n whereas the size of *TimeD* depends on n and the number of time periods in the planning horizon. At most $3n(n-1)$ Boolean variables, $3n(n-1)$ equivalence relations, $5(n-1)(n-2)/2$ redundant constraints and n sum constraints are generated. Another cumulative resource constraint can reuse the Boolean variables and requires only adding n new sum constraints.

Example 4.4. Consider the problem of Ex. 4.2 after the addition of $s_{\mathtt{c}} \leq 9$, $s_{\mathtt{e}} \leq 4$. The domains from precedence constraints are $D(s_{\mathtt{a}}) = [0..1]$, $D(s_{\mathtt{b}}) = [2..3]$, $D(s_{\mathtt{c}}) = [8..9]$, $D(s_{\mathtt{d}}) = [0..2]$, $D(s_{\mathtt{e}}) = [2..4]$, $D(s_{\mathtt{f}}) = [0..14]$. Propagation on the decomposition learns $\neg B^2_{\mathtt{ab}}$, $\neg B^2_{\mathtt{bc}}$ and $\neg B^2_{\mathtt{de}}$ directly from precedence constraints and hence $\neg B_{\mathtt{ab}}$, $\neg B_{\mathtt{bc}}$, and $\neg B_{\mathtt{de}}$. From the start times propagation determines that $B^1_{\mathtt{ab}}$, $B^1_{\mathtt{db}}$, $B^1_{\mathtt{ae}}$, $B^1_{\mathtt{de}}$, $B^1_{\mathtt{ac}}$, $B^1_{\mathtt{bc}}$, $B^1_{\mathtt{dc}}$, $B^1_{\mathtt{ec}}$, and similar facts about B^2_{**} variables, but no information about B_{**} variables. The sum constraints determine that $\neg B_{\mathtt{cf}}$ and $\neg B_{\mathtt{fc}}$, but there are no bounds changes. This illustrates the weaker propagation of the *ActiD* decomposition compared to the *TimeD* decomposition. □

If we use end time variables $e[i] = s[i] + p[i]$ instead of start time variables, we can generate a symmetric *ActiD* decomposition to that defined above.

In comparison to the *TimeD* decomposition, the *ActiD* decomposition is stronger in its ability to relate to activity precedence information ($i \ll j$), but generates a weaker profile of resource usage, since no implicit profile is recorded. They are thus incomparable in strength of propagation, although in practice the *TimeD* decomposition it almost always stronger.

4.5 Explanations for Cumulative Propagators

Propagators for `cumulative` check the consistency of the current CRP or prune inconsistent values from the variables domain. In order to gain maximum benefit from the underlying SAT solver (in terms of nogood learning) these algorithms must explain inconsistency and domain changes of variables using Boolean literals that encode the integer variables and domains in the SAT solver. When using LCG it

is not strictly necessary that a propagator explains all its propagations, in which case the resulting propagated constraints are treated like decisions, and learning is not nearly as useful. The decomposition approaches to `cumulative` inherit the ability to explain from the explanation capabilities of their base constraints. The challenge in building an explaining global constraint is to minimise the overhead of the explanation generation and make the explanations as reusable as possible.

In this section, explanations that only take variable start times into account are developed and they are then extended to explanation that can additionally consider flexible processing times, resource usages, and resource capacity, in the next section.

4.5.1 Consistency Check

Our cumulative propagator first performs a consistency check to determine whether the constraint is satisfiable. Here we consider the time-table consistency check based on the resource profile. If an overload in resource usage occurs on a resource with a maximal capacity R in the time interval $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$ involving the set of activities Ω, the following condition holds:

$$\forall i \in \Omega: \quad ub(s[i]) \leq \mathbf{s} \ \wedge \ \mathbf{e} \leq lb(s[i]) + p[i] \ \wedge \ \sum_{i \in \Omega} r[i] > R$$

Note that here and in the subsequent text $lb(.)$ and $ub(.)$ refers to the lower and the upper bound of the variable in the current state of propagation. A *naïve explanation* explains the inconsistency using the current domain bounds on the corresponding variables from the activities in Ω, which is always a correct explanation for any constraint.

$$\bigwedge_{i \in \Omega} [\![lb(s[i]) \leq s[i]]\!] \wedge [\![s[i] \leq ub(s[i])]\!] \rightarrow \mathit{false}$$

In some cases some activity in Ω might have compulsory parts before or after the overload. These parts are not related to the overload, and give us the possibility to widen the bounds in the explanation. We can widen the bounds of all activities in this way so that their compulsory part is only between $\mathbf{s}$ and $\mathbf{e}$. The corresponding explanation is called a *big-step explanation.*

$$\forall i \in \Omega: [\![\mathbf{e} - p[i] \leq s[i]]\!] \wedge [\![s[i] \leq \mathbf{s}]\!] \rightarrow \mathit{false}$$

The big-step explanation explains the reason for the overload over its entire width.

We can instead explain the overload by concentrating on a single time period t

in $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$ rather than examining the whole time interval. This allows us to strengthen the explanation. The explanation has the same pattern as a maximal explanation except we use t for $\mathbf{s}$ and $t+1$ for $\mathbf{e}$. We call these explanations *pointwise explanations*. The pointwise and big-step explanation coincide iff $\mathbf{s}+1=\mathbf{e}$.

$$\bigwedge_{i\in\Omega} [\![t - p[i] < s[i]]\!] \wedge [\![s[i] \leq t]\!] \rightarrow \mathit{false} \tag{4.3}$$

The pointwise explanation is implicitly related to the *TimeD* where an overload is explained for one time period. That time period by *TimeD* depends on the propagation order of the constraints related to an overload whereas for `cumulative` we have the possibility of choosing a time period to use to explain inconsistency. This means that we have more control and flexibility about what explanation is generated which may be beneficial. In the experiments reported herein we always choose the mid-point of $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$.

There are many open questions related to time periods for pointwise explanations in general and with respect to the possible search space reduction from derived nogoods or explanations: does it matter which time period is picked, if so, which is the best one?

Sometimes a resource overload is detected where the activities set Ω of activities that have a compulsory part at that time which is not minimal with respect to the resource capacity, *i.e.*, there exists a proper subset of activities $\Omega' \subset \Omega$ with $\sum_{i\in\Omega'} r[i] > c$. The same situation happens for the *TimeD* decomposition as well. Here, again with the global view of `cumulative`, we know the context of the activities involved and can decide which subset Ω' is used in order to explain the inconsistency if a choice exists. Here as well, it is an open question which subset is the best to restrict the search space most, or whether it does not matter? For our experiments the lexicographic least set of activities is chosen where the order is given by the order of appearance in `cumulative`, unless it is otherwise stated.

Example 4.5. Consider the problem of Ex. 4.1 with the additional constraints $s_{\mathtt{c}} \leq 9$, $s_{\mathtt{e}} \leq 4$, and $s_{\mathtt{f}} \leq 4$. The resulting bounds from precedence constraints are $D(s_{\mathtt{a}}) = [\,0\,..\,1\,]$, $D(s_{\mathtt{b}}) = [\,2\,..\,3\,]$, $D(s_{\mathtt{c}}) = [\,8\,..\,9\,]$, $D(s_{\mathtt{d}}) = [\,0\,..\,2\,]$, $D(s_{\mathtt{e}}) = [\,2\,..\,4\,]$, $D(s_{\mathtt{f}}) = [\,0\,..\,4\,]$. The time interval of positions where the activities can fit and the resulting resource profile are shown in Fig. 4.4. There is an overload of the resource capacity between time 4 and 6 with $\Omega = \{\mathtt{b},\mathtt{e},\mathtt{f}\}$., The simple explanation is $[\![2 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 3]\!] \wedge [\![2 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 4]\!] \wedge [\![0 \leq s_{\mathtt{f}}]\!] \wedge [\![s_{\mathtt{f}} \leq 4]\!] \rightarrow \mathit{false}$. The maximal explanation is $[\![0 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 4]\!] \wedge [\![1 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 4]\!] \wedge [\![0 \leq s_{\mathtt{f}}]\!] \wedge [\![s_{\mathtt{f}} \leq 4]\!] \rightarrow \mathit{false}$. A minimal explanation picking time 5 is $[\![-1 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 5]\!] \wedge [\![1 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq$

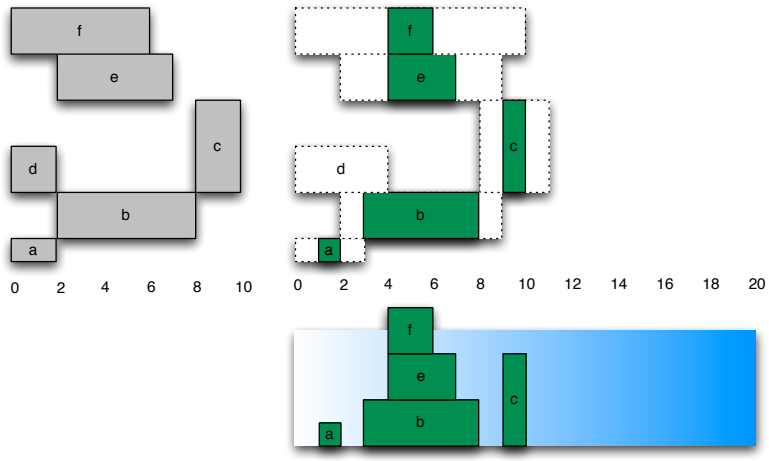


Figure 4.4: An example of an inconsistent partial schedule for the cumulative constraint.

$5[\![\wedge [\![-1 \leq s_{\mathtt{f}}]\!] \wedge [\![s_{\mathtt{f}} \leq 5]\!] \rightarrow \mathit{false}$. Note that each explanation is stronger than the previous one, and hence more reusable. Note also that some of the lower bounds (0 and -1) are universally true and can be omitted from the explanations. □

4.5.2 Time-Table Filtering

Time-table filtering is based on the resource profile of the compulsory parts of all activities. In a filtering without explanation the height of the compulsory parts concerning one time period or a time interval is given. For an activity the profile is scanned through to detect time intervals where it cannot be executed. The lower (upper) bound of the activity's start time is updated to the first (last) possible time period with respect to those time intervals. If we want to explain the new lower (upper) bound we need to know additionally which activities have the compulsory parts of those time intervals.

A *profile* is a triple (A, B, C) where $A = [\,\mathbf{s}\,..\,\mathbf{e}-1\,]$ is a time interval, B the set of all activities i with $ub(s[i]) \leq \mathbf{s}$ and $lb(s[i]) + p[i] \geq \mathbf{e}$ (that is a compulsory part in the time interval $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$), and C the sum of the resource usages $r[i]$ of all activities i in B. Here, we only consider profiles with a maximal time interval A with respect to B and C, *i.e.*, no other profile $([\,\mathbf{s}'\,..\,\mathbf{e}'-1\,], B, C)$ exists where $\mathbf{s}' = \mathbf{e}$ or $\mathbf{e}' = \mathbf{s}$.

Let us consider the case when the lower bound of the start time variable for activity j can be maximally increased from its current value $lb(s[j])$ to a new value $LB[j]$ using time-table filtering (the case of decreasing upper bounds is analogous and omitted). Then there exists a sequence of profiles $[D_1, \ldots, D_p]$ where $D_i =$

$([\,\mathbf{s}_i\,..\,\mathbf{e}_i-1\,],B_i,C_i)$ where $\mathbf{e}_0 = lb(s[j])$ and $\mathbf{e}_p = LB[j]$ such that

$$\forall 1 \leq i \leq p: \quad C_i + r[j] > R \ \wedge \ \mathbf{s}_i \leq \mathbf{e}_{i-1} + p[j]$$

Hence each profile D_i pushes the start time of activity j to $\mathbf{e}_i$.

A *naïve* explanation of the whole propagation would reflect the current domain bounds from the involved activities.

$$\left([\![lb(s[j]) \leq s[j]]\!] \wedge \bigwedge_{1 \leq i \leq p,\ k \in B_i} [\![lb(s[k]) \leq s[k]]\!] \wedge [\![s[k] \leq ub(s[k])]\!] \right) \rightarrow [\![LB[j] \leq s[j]]\!]$$

As for the consistency check it is possible to use smaller (bigger) values in the inequalities to get a big-step explanation.

$$\left([\![\mathbf{s}_1 + 1 - p[j] \leq s[j]]\!] \wedge \bigwedge_{1 \leq i \leq p,\ k \in B_i} [\![\mathbf{e}_i - p[k] \leq s[k]]\!] \wedge [\![s[k] \leq \mathbf{s}_i]\!] \right) \rightarrow [\![LB[j] \leq s[j]]\!]$$

Both the above explanations are likely to be very large—they involve all start times appearing in the sequence of profiles—and hence are not likely to be very reusable.

One solution is to generate separate explanations for each profile D_i starting from the earliest time interval. An explanation for the profile $D_i = ([\,\mathbf{s}_i\,..\,\mathbf{e}_i-1\,],B_i,C_i)$ which forces the lower bound of activity j to move from $\mathbf{e}_{i-1}$ to $\mathbf{e}_i$ is

$$\left([\![\mathbf{s}_i + 1 - p[j] \leq s[j]]\!] \wedge \bigwedge_{k \in B_i} [\![\mathbf{e}_i - p[k] \leq s[k]]\!] \wedge [\![s[k] \leq \mathbf{s}_i]\!] \right) \rightarrow [\![\mathbf{e}_i \leq s[j]]\!] \ ,$$

which corresponds to a big-step explanation of an inconsistency over the interval $[\,\mathbf{s}_i\,..\,\mathbf{e}_i-1\,]$.

Again we can use pointwise explanations based on single time periods rather than a big-step explanation for the whole time interval. Unlike the consistency case, we may need to pick a set of time periods no more than $p[j]$ apart to explain the increasing of the lower bound of $s[j]$ over the time interval. For a profile with length greater than the processing time of activity j we may need to pick more than one time period in a profile. Let $[t_1, \ldots, t_m]$ be a set of time periods such that $t_0 = lb(s[j])$, $t_m + 1 = LB[j]$, $\forall 1 \leq l \leq m : t_{l-1} + p[j] \geq t_l$ and there exists a mapping $P(t_l)$ of time periods to profiles such that $\forall 1 \leq l \leq m : \mathbf{s}_{P(t_l)} \leq t_l < \mathbf{e}_{P(t_l)}$ Then we

build a pointwise explanation for each time period t_l, $1 \leq l \leq m$

$$\left([\![t_l + 1 - p[j] \leq s[j]]\!] \wedge \bigwedge_{k \in B_i} [\![t_l + 1 - p[k] \leq s[k]]\!] \wedge [\![s[k] \leq t_l]\!] \right) \rightarrow [\![t_l + 1 \leq s[j]]\!] \tag{4.4}$$

This corresponds to a set of pointwise explanations of inconsistency. We use these pointwise explanations in our experiments, by starting from $t_0 = lb(s[j])$ and for $j \in [\,1\,..\,m\,]$ we choose t_j as the greatest time that maintains the conditions above. The exception is that we never entirely skip a profile D_i even if this is possible, but instead choose $\mathbf{e}_i - 1$ as the next time period and continue the process. Our experiments show this is slightly preferable to skipping a profile entirely.

Example 4.6. Consider the example shown in Fig. 4.3(b). which adds $s_{\mathtt{c}} \leq 9$, $s_{\mathtt{e}} \leq 4$ to the original problem. The profile filtering propagator can determine that activity `f` can start earliest at time 10, since it cannot fit earlier. Clearly because there is one resource unit missing—one available but two required—in the tine interval $[\,4\,..\,7\,]$ it must either end before or start after this interval. Since it cannot end before it must start after this time. Similarly for the time interval $[\,9\,..\,10\,]$ is must either end before or start after, and since it cannot end before it must start after. So for this example $lb(s_{\mathtt{f}}) = 0$ and $LB[\mathtt{f}] = 10$ and there are two profiles used $[D_1, D_2] = [([\,4\,..\,7\,], \{\mathtt{b}, \mathtt{e}\}, 4), ([\,9\,..\,10\,], \{\mathtt{c}\}, 4)]$.

The naïve explanation is just to take the bounds of the activities (`b`, `e`, `c`) involved in the profile that is used: e.g. $[\![2 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 3]\!] \wedge [\![2 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 4]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{f}} \leq 9]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$. (Note we omit redundant literals such as $[\![0 \leq s_{\mathtt{f}}]\!]$).

The iterative profile explanation explains each profile separately as $[\![1 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 4]\!] \wedge [\![2 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 4]\!] \rightarrow [\![7 \leq s_{\mathtt{f}}]\!]$ and $[\![4 \leq s_{\mathtt{f}}]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 9]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$.

The iterative pointwise explanation picks a set of time periods, say 5 and 9, whose corresponding profiles are D_1 and D_2, and explains each time period minimally giving: $[\![s_{\mathtt{b}} \leq 5]\!] \wedge [\![1 \leq s_{\mathtt{e}}]\!] \wedge [\![s_{\mathtt{e}} \leq 5]\!] \rightarrow [\![6 \leq s_{\mathtt{f}}]\!]$ and $[\![4 \leq s_{\mathtt{f}}]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 9]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$. Note that this explanation is analogous to the explanation devised by the decomposition in Ex. 4.3, and stronger than the iterative profile explanation. □

The global `cumulative` using time-table filtering and the *TimeD* decomposition have the same propagation strength. The advantage of the global approach is that we can control the times periods we propagate on, while the decomposition in the worst case may propagate on every time period in every profile. The possible advantage of the decomposition is that it learns smaller nogoods related to the decomposed variables, but since B_{it} simply represents a fixed conjunction of bounds in practice the nogoods learned by the *TimeD* decomposition have no advantage.

4.5.3 (Extended) Edge-Finding Filtering

The (extended) edge-finding filtering (Nuijten 1994) is based on activity intervals and reasoning about the energy of activities. Edge finding finds a set of activities Ω that all must occur in the time interval $[\,\mathbf{s}_\Omega \,..\, \mathbf{e}_\Omega\,]$ such that the total resources used by Ω are close to the amount available in that time interval ($R \times (\mathbf{e}_\Omega - \mathbf{s}_\Omega)$). If placing activity j at its earliest start time requires more than the remaining amount of resources from this range, then activity j cannot be strictly before any activity in Ω. We can then update its lower bound accordingly.

Since the edge-finding and extended edge-finding are highly related we combine them in one rule and refer to them just as edge-finding for simplicity. Given a set of activities Ω and activities $j \notin \Omega$ where $energy_\Omega$, $\mathbf{e}_\Omega$ and $\mathbf{s}_\Omega$ generalise the notation of the energy, the end time and start time from activities to activity sets, and o^j_Ω is the maximum resource usage for activity j before $\mathbf{s}_\Omega$:

$$energy_\emptyset = 0 \quad energy_\Omega = \sum_{i \in \Omega} energy_i \quad \mathbf{e}_\emptyset = +\infty \quad \mathbf{e}_\Omega = \max_{i \in \Omega}(ub(s[i]) + p[i])$$
$$\mathbf{s}_\emptyset = -\infty \quad \mathbf{s}_\Omega = \min_{i \in \Omega}(lb(s[i])) \quad o^j_\Omega = r[j] \times \max(\mathbf{s}_\Omega - lb(s[j]), 0)$$

then

$$j \in \mathcal{V}, \Omega \subseteq \mathcal{V} \setminus \{j\} : energy_\Omega + energy_j > R \times (\mathbf{e}_\Omega - \mathbf{s}_\Omega) + o^j_\Omega \Rightarrow j \not\ll i . \forall i \in \Omega \quad (4.5)$$

If the rule holds the lower bound of the start time of the activity j can be increased to

$$LB[j] := \max_{\Omega' \subseteq \Omega : rest(\Omega', r[j]) > 0} \mathbf{s}_{\Omega'} + \left\lceil \frac{rest(\Omega', r[j])}{r[j]} \right\rceil$$

where $rest(\Omega', r[j]) = energy_{\Omega'} - (R - r[j]) \times (\mathbf{e}_{\Omega'} - \mathbf{s}_{\Omega'})$. Figure 4.5 illustrates the (extended) edge-finding rule.

A naïve explanation would be

$$\left([\![lb(s[j]) \leq s[j]]\!] \wedge \bigwedge_{i \in \Omega} ([\![lb(s[i]) \leq s[i]]\!] \wedge [\![s[i] \leq ub(s[i])]\!]) \right) \rightarrow [\![LB[j] \leq s[j]]\!]$$

In order to gain a stronger explanation we can maximise the bounds on the activities in Ω. For that we have to consider that the condition for the edge-finding solely depends on Ω, j and the new lower bound for j on a subset Ω' of Ω. We obtain a

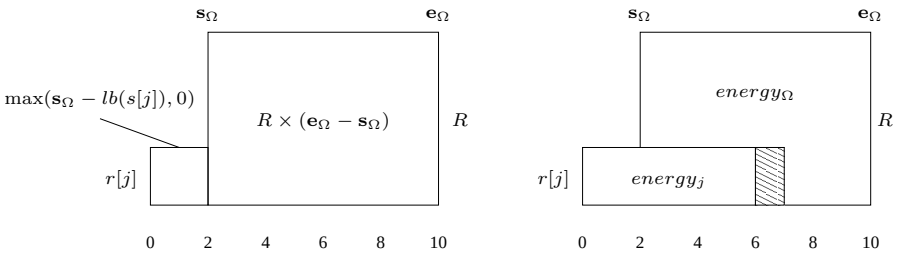


Figure 4.5: The left hand side of the figure illustrates the available energy within the interval $[\,\mathbf{s}_\Omega\,..\,\mathbf{e}_\Omega\,]$ plus the additional energy o^j_Ω when activity j starts earlier than $\mathbf{s}_\Omega$, while the right hand side illustrates the required energy if j starts earlier than all activities in Ω. For the illustrated situation we have $j = \mathtt{f}$, $lb(s[j]) = 0$ and $\Omega = \{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$. Since there is unused energy (the shaded area) no propagation occurs.

big-step explanation.

$$\left([\![\mathbf{s}_{\Omega\cup\{j\}} \leq s[j]]\!] \wedge \bigwedge_{i\in\Omega\setminus\Omega'} ([\![\mathbf{s}_\Omega \leq s[i]]\!] \wedge [\![s[i] + p[i] \leq \mathbf{e}_\Omega]\!]) \wedge \bigwedge_{i\in\Omega'} ([\![\mathbf{s}_{\Omega'} \leq s[i]]\!] \wedge [\![s[i] + p[i] \leq \mathbf{e}_{\Omega'}]\!]) \right) \rightarrow [\![LB[j] \leq s[j]]\!]$$

If we look at the unused activity of the time interval $[\,\mathbf{s}_\Omega\,..\,\mathbf{e}_\Omega\,]$ and the energy needed for the activity j in that time interval if j is scheduled at its earliest start time $lb(s[j])$ then the latter energy minus the former can be larger than 1. This would mean that the remaining energy can be used to strengthen the explanation in some way.

The remaining energy Δ concerning an activity j and the Ω satisfying the edge-finding condition is

$$\Delta = \mathbf{e}_\Omega + r[j] \times (lb(s[j]) + p[j] - \max(\mathbf{s}_\Omega, s[j])) - R \times (\mathbf{e}_\Omega - \mathbf{s}_\Omega)$$

Different (non-exclusive) options exist to use this energy to strengthen the explanation where $\Omega' \subseteq \Omega$ maximises $LB[j]$.

1. By increasing the value of the end time $\mathbf{e}_\Omega$ if $\Delta/c > 1$, but not for Ω'.
2. By decreasing the value of the start time $\mathbf{s}_\Omega$ if $\Delta/c > 1$, but not for Ω'.
3. By decreasing the value of the start time $\mathbf{s}_j$ if $\Delta/r[j] > 1$.
4. By removing an activity i from $\Omega \setminus \Omega'$ if $\Delta > energy_i$

Finally, if several $\Omega' \subseteq \Omega$ exist with which the lower bound of an activity j could be improved then the explanations can be split into several explanation as in the time-table filtering case. The filtering algorithm presented by Vilím (2009)[2] and Mercier and Van Hentenryck (2008) directly compute $LB[j]$ and its stepwise computation is hidden in the core of those algorithms. Hence, they have to be adjusted in a way that can increase the runtime complexity from $\mathcal{O}(kn\log(n))$ to $\mathcal{O}(kn^2\log(n))$ and $\mathcal{O}(kn^2)$ to $\mathcal{O}(kn^3)$ where n is the number of activities and k the number of distinct resource usages.

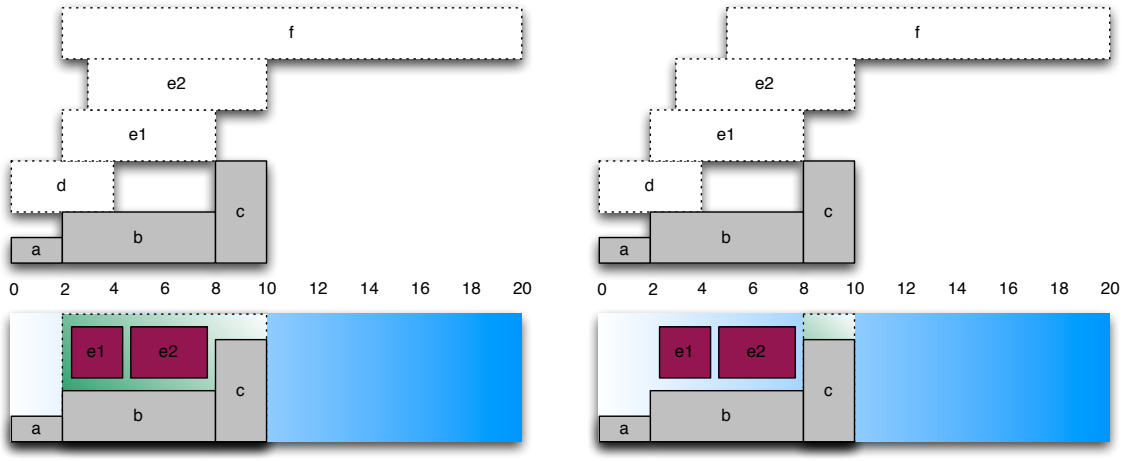
For a pointwise explanation, rather than consider all $\Omega' \subseteq \Omega$ we restrict ourselves to explain the bounds update generated by Ω. The big-step explanation is generated as before (but $\Omega = \Omega'$). With the new bound the edge-finding rule (4.5) will now hold for Ω' and we can explain that. Clearly, the pointwise explanation is stronger than the big-step explanation, because we broaden the bounds requirements on the activities in Ω' in the big-step explanation, and the later explanation only considers activities in Ω'.

Example 4.7. Consider the example of Ex. 4.1 where we split the activity `e` into two parts `e1` of duration 2 and `e2` of duration 3, but place no precedence constraints on these new activities. Suppose the domains of the variables are $D(s_{\mathtt{a}}) = \{0\}$, $D(s_{\mathtt{b}}) = \{2\}$, $D(s_{\mathtt{c}}) = \{8\}$, $D(s_{\mathtt{d}}) = [\,0\,..\,2\,]$, $D(s_{\mathtt{e1}}) = [\,2\,..\,6\,]$, $D(s_{\mathtt{e2}}) = [\,2\,..\,5\,]$, $D(s_{\mathtt{f}}) = [\,2\,..\,14\,]$, so activities `a`, `b` and `c` are fixed. The situation is illustrated in Fig. 4.6. The time-table filtering propagator cannot determine any propagation since `f` seems to fit at time 2, or after time 10. But in reality there are not enough resources for `f` in the time interval $[\,2\,..\,10\,]$ since this must include the activities `b`, `c`, `e1`, and `e2`. The total amount of resources available here is $5 \times 8 = 40$, but 12 are taken by `b` and 8 are taken by `c`, `e1` requires 4 resource units somewhere in this interval, and `e2` required 6 resource units. Hence there are only 10 resource units remaining in the interval $[\,2\,..\,10\,]$. Starting `f` at time 2 requires at least 12 resource units be used within this interval hence this is impossible.

The edge-finding condition holds for $\Omega = \{\mathtt{b}, \mathtt{c}, \mathtt{e1}, \mathtt{e2}\}$ and `f`. Now $rest(\{\mathtt{c}\}, 2) = 10 - (5-2) \times (10-8) = 4$ for $\Omega' = \{\mathtt{c}\}$. The lower bound calculated is $LB[\mathtt{f}] = 8 + \lceil 4/2 \rceil = 10$. The naïve explanation is $[\![2 \leq s_{\mathtt{f}}]\!] \wedge [\![2 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 2]\!] \wedge [\![2 \leq s_{\mathtt{e1}}]\!] \wedge [\![s_{\mathtt{e1}} \leq 6]\!] \wedge [\![2 \leq s_{\mathtt{e2}}]\!] \wedge [\![s_{\mathtt{e2}} \leq 5]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 8]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$.

The big-step explanation ensures that each activity in $\Omega \setminus \Omega'$ uses at least the amount of resources in the interval $[\,2\,..\,10\,]$ as found in the reasoning above. It is $[\![2 \leq s_{\mathtt{f}}]\!] \wedge [\![2 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 4]\!] \wedge [\![2 \leq s_{\mathtt{e1}}]\!] \wedge [\![s_{\mathtt{e1}} \leq 8]\!] \wedge [\![2 \leq s_{\mathtt{e2}}]\!] \wedge [\![s_{\mathtt{e2}} \leq 7]\!] \wedge [\![8 \leq$

[2] Vilím only presents the edge-finding algorithm, but the algorithm can be extended for extended edge-finding.



(a) (b)

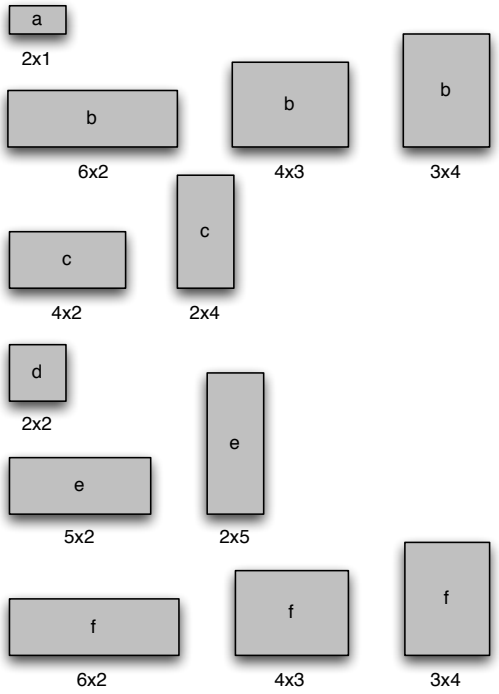
Figure 4.6: (a) An example of propagation of the cumulative constraint using edge-finding. (b) The result of propagating after the first step of stepwise edge-finding.

$s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 8]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$.

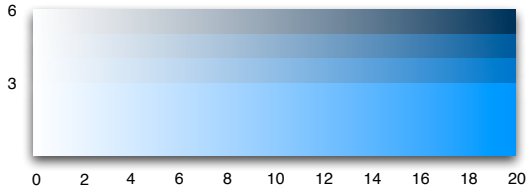
Now let us consider the pointwise explanation. If we restrict ourselves to use Ω for calculating the new lower bound we determine $LB[j] = 2 + \lceil (30 - (5-2) \times (10-2))/2 \rceil = 5$ and the big-step explanation is $[\![2 \leq s_{\mathtt{f}}]\!] \wedge [\![2 \leq s_{\mathtt{b}}]\!] \wedge [\![s_{\mathtt{b}} \leq 4]\!] \wedge [\![2 \leq s_{\mathtt{e1}}]\!] \wedge [\![s_{\mathtt{e1}} \leq 8]\!] \wedge [\![2 \leq s_{\mathtt{e2}}]\!] \wedge [\![s_{\mathtt{e2}} \leq 7]\!] \wedge [\![2 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 8]\!] \rightarrow [\![5 \leq s_{\mathtt{f}}]\!]$, which results in the situation shown in Fig. 4.6(b). We then detect that edge-finding condition now holds for $\Omega' = \{\mathtt{c}\}$ which creates a new lower bound 10, and explanation $[\![5 \leq s_{\mathtt{f}}]\!] \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 8]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$. The original big-step explanation is broken into two parts, each logically stronger than the original.

For the original big-step explanation $\Delta = 30 + 2 \times (2+6-2) - 5 \times (10-2) = 2$. We cannot use this to improve the explanation since it is not large enough. But for the second explanation in the pointwise approach $\Delta = 10 + 2 \times (5+6-8) - 5 \times (10-8) = 6$. We could weaken the second explanation (corresponding the period 3 in the enumeration on the page 93) to $\underline{[\![3 \leq s_{\mathtt{f}}]\!]} \wedge [\![8 \leq s_{\mathtt{c}}]\!] \wedge [\![s_{\mathtt{c}} \leq 8]\!] \rightarrow [\![10 \leq s_{\mathtt{f}}]\!]$ because $\Delta / r[\mathtt{f}] = 6/2 = 3 > 1$. □

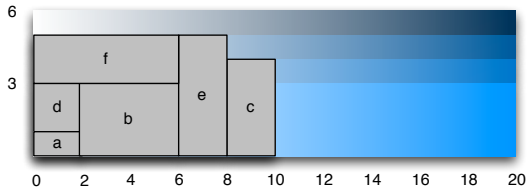
We have not yet implemented edge-finding with explanation, since the problems we examine are not highly cumulative and for such problems edge-finding filtering cannot compete with time-table filtering. It remains interesting future work to experimentally compare different approaches to edge-finding filtering with explanation.



(a) All possible combinations.



(b) Variable resource capacity R.



(c) Optimal schedule for $R = 5$.

Figure 4.7: (a) shows all possible combinations for the processing time and resource usages; (b) shows the resource R with the flexible resource capacity between 3 and 6; and (c) an optimal schedule with respect to $R = 5$.

4.6 Explanation Extensions for Cumulative Propagators

In the previous section explanations are developed for the most common case of the cumulative constraint where only start times are variable, *i.e.*, all other parameters of the activities are fixed. These explanations have different strengths, where the explanations based on pointwise explanation are the strongest ones. These strongest explanations are extended in this section, so that they can take variable processing times, resource usages, and resource capacities into account.

Example 4.8. Consider the example of Ex. 4.1. We assume that the processing times and resource usages for the activities `b`, `c`, `e`, and `f` are flexible but constrained to the fixed energies 12, 8, 10, and 12 respectively and that resource usages are greater than 1. Moreover, we assume that the resource capacity R can take any value from 3 to 6. Fig. 4.7 shows all combinations of processing time and resource usage for each activity, the variable resource capacity, and an optimal schedule in the case of $R = 5$. □

4.6.1 Time-Table Consistency Check

Suppose the activities in Ω overload the resource with resource capacity R in the time interval $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$ with respect to their current domain of the resource usages and the current domain of the resource capacity, *i.e.*, $\sum_{i\in\Omega} lb(r[i]) > ub(R)$. Then, the pointwise explanation describes the overload for one time period t in $[\,\mathbf{s}\,..\,\mathbf{e}-1\,]$. In the case that durations, resource usages, and the resource capacity are fixed, Eq. (4.3) on page 88 shows the corresponding explanation. Note that this explanation would be invalid if durations, resource usages or the resource capacity were variable.

The literal $[\![t-p[i] < s[i]]\!]$ in Eq. (4.3) expresses the necessary condition that the earliest end time $s[i]+p[i]$ must be greater than t in order to create a compulsory part at the time period t. If the duration is variable then $t-p[i] < s[i]$ becomes an inequality on two variables which cannot be represented by a domain literal in Lcg. Hence, we split it into two parts that each contain only one variable, which allows us to use the literals from the variable domain encoding.

One possibility to split it into two inequalities is $t-lb(p[i]) < s[i]$ and $lb(p[i]) \leq p[i]$ which leads to the simple explanation $[\![t-lb(p[i]) < s[i]]\!] \wedge [\![lb(p[i]) \leq p[i]]\!]$. Instead of using the constants $lb(p[i])$ in both inequalities any value in $[\,t-lb(s[i])+1\,..\,lb(p[i])\,]$ would be valid. The bounds for this range results from the fact that only for these values the corresponding literals $[\![t-lb(p[i]) < s[i]]\!]$ and $[\![lb(p[i]) \leq p[i]]\!]$ are *true* at the same time with respect to the current domain of $s[i]$ and $p[i]$—this is necessary for creating an unit explanation. Later in this section we discuss which values from the range should be picked or not.

Beside the flexible processing times it is also possible that resource usages and the resource capacity are variables. The propagator has to explain the resource overload created by the activities in Ω at the time period t regarding the domain of resource usages and resource capacity. A simple explanation takes the lower bound of $r[i]$ for each $i \in \Omega$ and the upper bound of R into account:

$$[\![R \leq ub(R)]\!] \wedge \bigwedge_{i\in\Omega} [\![lb(r[i]) \leq r[i]]\!] \ .$$

This may not result in the strongest explanation if $\sum_{i\in\Omega} lb(r[i]) > ub(R)+1$. For instance, a removal of an activity $j \in \Omega$ does not resolve the conflict if $\sum_{i\in\Omega} lb(r[i]) - ub(R) - 1 > lb(r[j])$. How we can strengthen it is discussed later in the section.

Combining the simple explanations with variable durations, resource usages and the resource capacity into the pointwise explanation leads to the following explana-

tion

$$\left(\bigwedge_{i\in\Omega}[\![t - lb(p[i]) < s[i]]\!] \wedge [\![lb(p[i]) \leq p[i]]\!] \wedge [\![s[i] \leq t]\!]\right) \wedge [\![R \leq ub(R)]\!] \wedge \left(\bigwedge_{i\in\Omega}[\![lb(r[i]) \leq r[i]]\!]\right) \rightarrow \mathit{false} \ , \quad (4.6)$$

where the first part explains the creation of compulsory parts at the time period t and the second part the resource overload.

4.6.2 Time-Table Filtering

Suppose for the activity j the lower bound of its start time variable can be updated to $LB[j]$. This update is explained by a set of pointwise explanations covering a sequence of increasing time periods $t_1, \ldots, t_m$ such that $t_0 = lb(s[j])$, $t_m + 1 = LB[j]$, and $t_{l-1} + p[j] \geq t_l$ for all $l \in [\,1\,..\,m\,]$. In the remainder of this section we consider one of these time periods and simply denote it by t. If only the start times are variable then (4.4) shows the corresponding explanation. As for the consistency check this explanation is not valid if processing times, resource usages, or the resource capacity are variable. In the remainder of this section let t and Ω be one of these time periods in $t_1, \ldots, t_m$ and the corresponding set of activities such that $lb(r[j]) + \sum_{i\in\Omega} lb(r[i]) > ub(R)$.

In comparison to the consistency check the explanation only describes a possible resource overload at time period t if the activity j is executed at this period. Thus, the same explanations in (4.6) for the consistency check can be used for the activity j, the activities in Ω, and the resource capacity R to describe the possible resource overload. The remaining literals for the pointwise explanation are related to the variables $s[j]$, and $p[j]$.

The literal $[\![t - p[j] < s[j]]\!]$ in (4.4) describes a necessary condition for the execution of j at the period t, whereas the literal $[\![t < s[j]]\!]$ in (4.4) describes a sufficient condition for a non-execution of j at this period if the first literal is true. We simply split the first literal into $[\![t - lb(p[j]) < s[j]]\!]$ and $[\![lb(p[j]) \leq p[j]]\!]$ as we have done for the consistency check case, and the literal remains unchanged. Combining all explanations results in a pointwise explanation that considers variable start times,

processing times, resource usages, and resource capacities:

$$[\![t - lb(p[j]) < s[j]]\!] \wedge [\![lb(p[j]) \leq p[j]]\!] \wedge \left(\bigwedge_{i \in \Omega} [\![t - lb(p[i]) < s[i]]\!] \wedge [\![lb(p[i]) \leq p[i]]\!] \wedge [\![s[i] \leq t]\!] \right) \wedge [\![R \leq ub(R)]\!] \wedge \left(\bigwedge_{i \in \Omega \cup \{j\}} [\![lb(r[i]) \leq r[i]]\!] \right) \rightarrow [\![t < s[j]]\!] \ . \quad (4.7)$$

This explanation can be strengthened in the same way as for the consistency check which we explain in the remainder of this section.

4.6.3 Strengthening of Explanations for Time-Table Algorithms

We have presented pointwise explanations for the consistency check (see (4.6)) and the filtering algorithm (see (4.7)). We have pointed to two aspects of the explanation that can be strengthened or where we have options for different explanations. Here, we seek the strongest explanations, since they prune larger parts of the search space, and if multiple strongest explanations exist we look for the one that may propagate most frequently in the remaining search. The last decision depends on the search algorithm used and how it interacts with the conflict analysis.

One part of the explanation encodes the necessary condition for an activity $i \in \Omega$ to create a compulsory part at the time period t. This is done by the literals $[\![t - lb(p[i]) < s[i]]\!]$ and $[\![lb(p[i]) \leq p[i]]\!]$. Instead of using $lb(p[i])$ as constant in both inequalities, we can use any value q in $[\, t - lb(s[i]) + 1 \,..\, lb(p[i]) \,]$. But not all of them lead to a strongest explanation.

Example 4.9. Consider Ex. 4.8. For activity e the processing time p_e can take either 10, 5, or 2, and the resource usage r_e either 1, 2, or 5. Suppose $lb(s_\mathsf{e}) = 2$, and $lb(p_\mathsf{e}) = 5$ then q is either 4 or 5. Hence, we can explain the necessary condition either with $[\![1 < s_\mathsf{e}]\!] \wedge [\![4 \leq p_\mathsf{e}]\!]$ or $[\![0 < s_\mathsf{e}]\!] \wedge [\![5 \leq p_\mathsf{e}]\!]$ if $t = 5$. Since $p[\mathsf{e}]$ cannot take value 4, but 5, it follows $[\![4 \leq p_\mathsf{e}]\!] \rightarrow [\![5 \leq p_\mathsf{e}]\!]$. Because of this and $[\![0 < s_\mathsf{e}]\!] \nrightarrow [\![1 < s_\mathsf{e}]\!]$ the first explanation implies the second one, but not vice versa. Therefore, the second explanation is strictly stronger then the first one. □

The following proposition gives the characterisation for which values of q the explanation results in a strongest explanation.

Proposition 4.1. *Suppose an activity i creates compulsory parts with respect to the current domains of its start and processing time variable,* i.e., *$ub(s[i]) < lb(s[i]) + lb(p[i])$. Let $I_{s[i]}$ and $I_{p[i]}$ be sets of integers that $s[i]$ and $p[i]$ can take respectively. Moreover, let t be the time period in $[\, ub(s[i]) \,..\, lb(s[i]) + lb(p[i]) - 1\,]$ for which a compulsory part must be explained, and $I_{s[i]}(t)$ be $\{t - i + 1 \mid i \in I_{s[i]}\}$. Let $K = k_1, k_2, \ldots, k_m$ be the numbers in $Q \cap \big(I_{s[i]}(t) \cup I_{p[i]}\big)$ in increasing order, where $Q = [\, t - lb(s[i]) + 1 \,..\, lb(p[i])\,]$. Then the following set $Q(i,t)$ contains all integers q from K that result in a strongest explanation:*

$$
\begin{aligned}
Q(i,t) = \{k_j \mid 1 \le j < m, (k_j, k_{j+1}) \notin I^2_{s[i]}(t)\} \cup \\
\{k_j \mid k_j \in I_{s[i]}(t) \cap I_{p[i]}\} \cup \{k_m \mid k_m \in I_{s[i]}(t)\} \cup \\
\{k_j \mid 1 < j \le m, (k_j, k_{j-1}) \notin I^2_{p[i]}\} \cup \{k_1 \mid k_1 \in I_{p[i]}\} \ .
\end{aligned}
$$

Proof. Recall that an explanation C_1 is stronger than an explanation C_2 if $C_2 \rightarrow C_1$. Thus, C_1 is strictly stronger than C_2 if $C_1 \nrightarrow C_2$ holds additionally. We say C_2 is (strictly) weaker than C_1, respectively.

We prove the proposition in two steps. First, we show that for all values in $Q(i,t)$ the corresponding explanation is not strictly weaker than any other explanation. Second, we show that for all values in $K \setminus Q(i,t)$ the corresponding explanation is strictly weaker than at least one other explanation.
First, we have to prove the following:

$$
\forall q \in Q(i,t), \forall q' \in K \setminus \{q\} : expl(q) \nrightarrow expl(q') \text{ or } expl(q) \leftrightarrow expl(q') \ ,
$$

where $expl(q) \equiv [\![t - q < s[i]]\!] \wedge [\![q \le p[i]]\!]$.

Let q and q' be any number in $Q(i,t)$ and $K \setminus \{q\}$, respectively. Assume that $q = k_j$. We differentiate between the two cases: $k_j < q'$ and $k_j > q'$.

$(k_j < q')$: It follows $t - k_j < s[i] \rightarrow t - q' < s[i]$. Let j' be the smallest integer that satisfies $j \le j' \le m$ and $k_{j'} \in I_{p[i]}$. If $q' \ge k_{j'}$ then $k_j \le p[i] \nrightarrow q' \le p[i]$. Hence, $expl(k_j) \nrightarrow expl(q')$. Suppose $q' < k_{j'}$ now. Then $k_j \notin I_{p[i]}$ and it follows $k_j \le p[i] \rightarrow q' \le p[i]$, *i.e.*, $expl(k_j) \rightarrow expl(q')$. Since $k_j \in Q(i,t)$ can only come from the subset $\{k_l \mid 1 \le l < m, (k_l, k_{l+1}) \notin I^2_{s[i]}(t)\}$, it holds $j' = j + 1$ and therefore $t - q' < s[i] \rightarrow t - k_j < s[i]$. Together with $q' \le p[i] \rightarrow k_j \le p[i]$, it follows $expl(q') \rightarrow expl(k_j)$. Thus, this case is proven.

$(k_j > q')$: This case is symmetric to the previous case, just $s[i]$ and $[p[i]$ are swapped in the argument.

Second, we prove the following:

$$\forall q \in K \setminus Q(i,t), \exists q' \in K : expl(q) \rightarrow expl(q') \text{ and } expl(q') \not\rightarrow expl(q) \ .$$

Let q be any number in $K \setminus Q(i,t)$. Assume $q = k_j$ then $1 < j < m$ due to the definition of $Q(i,t)$. Moreover, k_j must be in either $I_{s[i]}(t)$ or $I_{p[i]}$, otherwise it would be in $Q(i,t)$. Now, we separately prove each case.

($k_j \in I_{s[i]}(t)$): Then, it holds that $k_j \leq p[i] \rightarrow k_{j+1} \leq p[i]$, since all $l \in [\, k_j \,..\, k_{j+1} - 1\,]$ are not in $I_{p[i]}$. Due to $t - k_j < s[i] \rightarrow t - k_{j+1} < s[i]$ it follows $expl(k_j) \rightarrow expl(k_{j+1})$. Now, we have to show $expl(k_{j+1}) \nrightarrow expl(k_j)$. Because of the definition of $Q(i,t)$ the value k_{j+1} must be in $I_{s[i]}(t)$, otherwise k_j would be in $Q(i,t)$. Thus, $t - k_{j+1} < s[i] \nrightarrow t - k_j < s[i]$, *i.e.*, $expl(k_{j+1}) \nrightarrow expl(k_j)$.

($k_j \in I_{p[i]}$): The proof is symmetric to the previous case.

Therefore, the proposition holds. □

In the proposition we do not consider a value in Q that belongs to neither $I_{s[i]}$ nor $I_{p[i]}$. These values either do not result in a strongest explanation or lead to an explanation that is equivalent to a strongest explanation.

It is desirable to choose a value in $Q(i,t)$, but the propagator might only have a partial knowledge about the sets $I_{s[i]}$ and $I_{p[i]}$, for example, if $p[i]$ takes two non-consecutive values and its domain is encoded as range. Hence, the propagator may pick a value between those values. Therefore, it might not always be possible for the propagator to create a strongest explanation.

An LCG solver encodes domains as ranges and provides to a propagator both the initial and current bounds. Moreover, LCG variables can be defined as variable views (Schulte and Tack 2005) on another LCG variable, *i.e.*, $y = ax + b$ where the variable y is a view on the variable x, and a, b are constants. Note that all variables in the model are internally represented as variable views, and these variables that are not defined as a variable view are represented as a view on themselves with $a = 1$, $x = y$, and $b = 0$. An LCG propagator has access to x, a, and b when consulting y. Thus, if the sets $I_{s[i]}$ and $I_{p[i]}$ can be described as variable views and the variables are defined $s[i]$ and $p[i]$ as these views then an LCG solver can deduce the set $Q(i,t)$.

In our implementation the explanations (4.6) and (4.7) uses $\max_{q \in Q(i,t)} q$ for $i \in \Omega$

and $\min_{q \in Q(j,t)} q$ for j with respect to these sets

$$I_{s[i]} = \{a_{s[i]} \cdot k + b_{s[i]} \mid k \in D_{init}(x_{s[i]})\} \text{ and}$$
$$I_{p[i]} = \{a_{p[i]} \cdot k + b_{p[i]} \mid k \in D_{init}(x_{p[i]})\} \ ,$$

where $s[i]$ and $p[i]$ are defined as variable views $s[i] = a_{s[i]} \cdot x_{s[i]} + b_{s[i]}$ and $p[i] = a_{p[i]} \cdot x_{p[i]} + b_{p[i]}$, respectively.

Another part of the explanations describes a resource overload at the time period t in (4.6) or a possible resource overload in (4.7) if the activity j would be executed at t. Let Ω' be the set of activities Ω for the first explanation and $\Omega \cup \{j\}$ for the second explanation. Thus, in both explanations the situation is described as

$$[\![R \leq ub(R)]\!] \wedge \bigwedge_{i \in \Omega'} [\![lb(r[i]) \leq r[i]]\!] \ .$$

This explanation may not result in a strongest explanation if $\sum_{i \in \Omega'} lb(r[i]) > ub(R) + 1$. The number $\mathit{lift} = \sum_{i \in \Omega'} lb(r[i]) - ub(R) - 1$ reflects the number of resource units that can be added to the resource capacity without resolving the overload. This means that we can use those units to strengthen the explanation. Here, strengthening means increasing q_R in $[\![R \leq q_R]\!]$ from $ub(R)$ to maximal $\min(\max_{D_{init}} R, ub(R) + \mathit{lift})$, or decreasing q_i in $[\![q_i \leq r[i]]\!]$ from $lb(r[i])$ to minimal $\max(\min_{D_{init}} r[i], lb(r[i]) - \mathit{lift})$ or 0 if $lb(r[i]) - \mathit{lift} \leq 0$ for all $i \in \Omega'$. A reduction of q_i to 0 represents a removal of i from the conflict, *i.e.*, removal of all related literals from the explanation.

All options are non-exclusive and which one has the most benefit is an open question. The answer is tightly correlated with the conflict analysis used and the search algorithm. For instance, if the goal is to minimise the resource capacity R and the search already proved a solution for $ub(R) + 1$ then to choose any value greater than $ub(R)$ for q_R is wasteful.

Here again, not every value for q_R and q_i leads to a strongest explanation. The following proposition characterises which combination of values results in a strongest explanation.

Proposition 4.2. *Let I_R and $I_{r[i]}$ be the set of integers that the variable R and $r[i]$ can respectively take for each $i \in \Omega'$. Moreover, let $I^0_{r[i]}$ be defined as $I_{r[i]} \cup \{0\}$. Then, Q_R and $Q_{r[i]}$ are defined as follows:*

$$Q_R = I_R \cap [\, ub(R) \,..\, \min(\max_{D_{init}} R, ub(R) + z)\,] \ ,$$
$$\forall i \in \Omega' : \quad Q_{r[i]} = I^0_{r[i]} \cap [\, \max(l(i), lb(r[i]) - \mathit{lift}) \,..\, lb(r[i])\,] \ ,$$

where $l(i) = \min_{D_{init}} r[i]$ *if* $lb(r[i]) - lift > 0$, *or* $l(i) = 0$ *otherwise. Furthermore, let* Γ *be a mapping that maps* q_R *and* q_i *to a value from their corresponding set* Q_R *and* $Q_{r[i]}$ *respectively. The mapping results in a valid strongest explanation if the following holds:*

$$\begin{aligned}
& \sum_{i \in \Omega'} (lb(r[i]) - \Gamma(q_i)) + \Gamma(q_R) - ub(R) \leq lift && (validity) \\
\forall i \in \Omega' : \quad & Q_{r[i]} \cap [\, \Gamma(q_i) - rest \,..\, \Gamma(q_i) - 1 \,] = \emptyset && (strength) \\
& Q_R \cap [\, \Gamma(q_R) + 1 \,..\, \Gamma(q_R) + rest \,] = \emptyset \ , && (strength)
\end{aligned}$$

where $rest = \max(0, lift - \sum_{i \in \Omega'} (lb(r[i]) - \Gamma(q_i)) + \Gamma(q_R) - ub(R))$.

Proof. The correctness of the proposition is straightforward and it is proven by contradiction. Suppose Γ is a mapping leading to a valid strongest explanation, but does not satisfy all equations. If the first equation does not hold for Γ then the corresponding explanation does not describe a resource overload, since $lift = \sum_{i \in \Omega'} lb(r[i]) - ub(R) - 1$. Hence, Γ would be not valid. If the second equation is not satisfied for $i \in \Omega'$ then every mapping Γ' results in a strictly stronger explanation than for Γ, where Γ' is defined by $\Gamma'(x) = \Gamma(x)$ for all $x \in \{q_R\} \cup \{q_j \mid j \in \Omega \wedge j \neq i\}$ and $\Gamma(q_i) = c$ where $c \in Q_{r[i]} \cap [\, \Gamma(q_i) - rest \,..\, \Gamma(q_i) - 1 \,]$. Thus, Γ would not result in a strongest explanation. The same argument holds for the remaining equation. Therefore, it is shown that Γ does not establish a valid strongest explanation. □

As mentioned before the propagator might only have partial knowledge about the sets I_R and $I_{r[i]}$, so that it may choose values that do not lead to a strongest explanation. Our implementation uses the following sets to generate the sets $Q_{r[i]}$ and Q_R:

$$\begin{aligned}
\forall i \in \Omega' \qquad & I_{r[i]} = \{a_{r[i]} \cdot k + b_{r[i]} \mid k \in D_{init}(x_{r[i]})\} \ , \text{and} \\
& I_R = \{a_R \cdot k + b_R \mid k \in D_{init}(x_R)\} \ ,
\end{aligned}$$

where $r[i]$ and R are defined as variable views $r[i] = a_{R[i]} \cdot x_{r[i]} + b_{r[i]}$ and $R = a_R \cdot x_R + b_R$, respectively.

Our implementation strengthens the explanation by using the available units *lift* as follows where *lift*′ are the remaining units from *lift* and the activities in $\Omega' \setminus \{j\}$ are tried out in input order.

1. $q_j = \min\{k \in Q_{r[j]} \mid lb(r[j]) - k \leq lift'\}$.

2. $q_R = \max\{k \in Q_R \mid k - ub(R) \leq \mathit{lift}'\}$.

3. $q_i = \min\{k \in Q_{r[i]} \mid lb(r[i]) - k \leq \mathit{lift}'\}$ for all $i \in \Omega' \setminus \{j\}$. If $q_i = 0$ then all literals related to the activity i are removed.

Overall, the implemented cumulative propagator generates a strongest explanation for (4.6) and (4.7) if it knows about the sets $I_{s[i]}$, $I_{p[i]}$, $I_{r[i]}$, and I_R. In other cases, it might not create a strongest explanation. Furthermore, during the progress of search the conflict analysis can infer that variables cannot globally take some values from their domain. This information could be used to pick the strongest explanation that does not include these "forbidden" values. This information is not available to an LCG propagator.

4.7 Final Remarks

In this chapter, we focussed on explanations for the cumulative propagation in an LCG solver. We first considered the *TimeD* and *ActiD* decomposition, the time-table consistency check and filtering, and the (extended) edge-finding filtering where only start times are variable. Then we extended the explanation for the two time-table algorithms, so that they can also take variable processing times, resource usages, and resource capacities into account.

First, we considered explanations created by two decompositions *TimeD* and *ActiD*. Those explanations are "fine-grained" and explain a resource overload or a propagation for one time period.

Second, we built an explaining global cumulative that generates pointwise and strongest explanations if sufficient information about valid values of variables is available to the propagator. Pointwise explanations only explain a sufficient part of a conflict or propagation step, *i.e.*, are smaller than explanations using the current domain bounds. In order to get the most benefit, *i.e.*, biggest reduction of the search space, a pointwise explanation must be strengthened.

During the progress of search, conflict analysis can infer values in the domains of variables that cannot take part in any solution of the problem with respect to the initial set of constraints and the set of nogoods. Thus, it does not seem beneficial to generate an explanation including those values. The same thing happens for values in the domain of the objective variable when the problem is optimised. The cumulative propagator in the LCG solver has no information about global invalid values in the domains of variables. Hence, it may create explanations which include values that cannot take part in any solution.

Furthermore, the global view on the cumulative constraint gives more control about how to strengthen the explanations and which resource conflict should be explained, than in the decomposition case. For instance, one can think about creating different explanations for different problem classes. But this needs further investigation.

5

Experiments on Resource-Constrained Project Scheduling Problems

The resource-constrained project scheduling problem (Rcpsp) is a basic scheduling problem involving multiple scarce cumulative resources, activities consuming one or more resources, and precedence relations between activities. A precedence relation expresses that one activity must have finished its execution before the other one can be started.

One important extension of Rcpsp generalises the precedence relations. These generalised precedence relations can express that an activity must start at least or at most a specific time after the other one has started. This problem is called resource-constrained project scheduling problem with generalised precedence relations (Rcpsp/max).

In this chapter, we evaluate the explanations for the cumulative constraint developed in the previous chapter on standard benchmarks for Rcpsp and Rcpsp/max. We show that our generic approaches outperform the state-of-the-art approaches—mostly problem specific—on these benchmarks.

5.1 Introduction

In Chap. 4, we introduced the cumulative resource scheduling (Crp) problem consisting of one renewable resource and non-preemptive activities requiring the resource for their executions. Rcpsp and Rcpsp/max are generalisations of this

problem. Both involve multiple renewable resources, non-preemptive activities consuming some of these resources, and (generalised) precedence relations between activities. The goal is to find a schedule of the activities that minimises the project duration, *i.e.*, the completion time of the last activity, and satisfies all resource and (generalised) precedence constraints. RCPSP and RCPSP/max are respective denoted as $PS|prec|C_{max}$ and $PS|temp|C_{max}$ by Brucker *et al.* (1999). Both problems are NP-hard (Błażewicz *et al.* 1983).

Here, the resources have a constant resource capacity over the planning horizon. Activities are characterised by variable start times, fixed processing times, and resource usages. Precedence relations between two activities i and j are expressed as $s_i + p_i \leq s_j$ where s_i and s_j respectively are the start times of i and j, and p_i the processing time of i. It is denoted by $i \ll j$. *Generalised precedence relations*[1] have the form $s_i + d_{ij} \leq s_j$ where d_{ij} is a discrete distance between them. If $d_{ij} \geq 0$ this imposes a *minimal time lag*, while if $d_{ij} < 0$ this imposes a *maximal time lag* between start times. These generalised precedence constraints are a subclass of UTVPI constraints discussed in Chap. 3 on page 45.

Generalised precedence constraints are significantly different from precedence constraints, since they cause the NP-hardness of deciding whether an RCPSP/max instance is feasible given an infinite planning horizon (Bartusch *et al.* 1988). The feasibility is solvable in polynomial time for RCPSP instances. The reason for this is the absence of maximal time lags, *i.e.*, here activity executions can always be delayed until a time period where enough resource units are available without breaking any precedence constraints.

However, practical scheduling problems can include substantially varied restrictions on the resources and activities. The following restrictions can be modelled with generalised precedence relations: minimal and maximal overload of activities, synchronisation of start or end times for activities, change of the resource usage during the activity's execution, fixed start times of activities, setup times, or non-delay execution of activities (see, e.g. Bartusch *et al.* 1988, Neumann and Schwindt 1997, Dorndorf *et al.* 2000).

In this chapter, we first present related work. We then devote ourselves to RCPSP problems. We show how decomposition of `cumulative` can be competitive with state-of-the-art specialised methods from the CP and OR community. We then show that building a global cumulative propagator with specialised explanation capabilities can further improve upon the explaining decompositions. After this we

[1] They are also called as temporal precedence relations, arbitrary precedence relations, minimal and maximal time lags, and time windows.

switch our focus to RCPSP/max problems and show how our complete approach including the global cumulative propagator with explanation, outperforms other complete methods in terms of the runtime and the quality of solutions. Surprisingly, it also outperforms the published incomplete approaches on the considered benchmarks in terms of the quality of solutions. The G12 Constraint Programming Platform (Stuckey *et al.* 2005) is used for implementation of the decomposed and global cumulative constraint as an LCG propagator. We evaluate our approach on RCPSP and RCPSP/max from the well-established and challenging benchmark library PSPLib (Kolisch and Sprecher 1997) and PSP/max-library (Schwindt 2011) which are accessible via `http://129.187.106.231/psplib/`.

5.2 Related Work

In this section we briefly present the state-of-the-art solvers from the AI, CP, and OR communities that are compared to our approach. These solvers are either competitive or the best solver on some parts of the considered benchmarks. Most of the solvers share the fact that they are problem specific, and not generic, as is our approach implemented in an LCG solver.

5.2.1 RCPSP

RCPSP is exhaustively studied in the literature and usually tested on benchmarks from the PSPLib: J30, J60, J90, and J120, where the numbers indicate the number of activities in each instance. These test sets were generated by ProGen (Kolisch *et al.* 1995). Recent surveys about complete and incomplete methods can be found in e.g. Artigues *et al.* (2008), Kolisch and Hartmann (2006), Demeulemeester and Herroelen (2002). These methods come from different areas such as dynamic programming, artificial intelligence, operations research, etc. (Kolisch 1996).

Due to the nature of RCPSP, *i.e.*, creation of a feasible schedule in polynomial time without restriction on the planning horizon, many heuristic approaches are based on serial and parallel scheduling generation schemes (SGS) (see e.g. Kelley 1963, Bedworth and Bailey 1982)[2] that stepwise assign start times to unscheduled activities. The serial and parallel SGS are deterministic algorithms that incrementally extend a partial schedule by choosing one or more eligible activities—*i.e.*, all of whose predecessors are fixed in the partial schedule—which is then scheduled. These schemes are combined with e.g. priority rules, or meta-heuristics. For more

[2]The parallel SGS is referred as Brooks' algorithm in Bedworth and Bailey (1982).

details about SGS, different methods based on them, and computational results in OR see e.g. Hartmann and Kolisch (2000), Kolisch (1996), Kolisch and Hartmann (2006).

The best known complete algorithm for solving RCPSP is from Demeulemeester and Herroelen (1992, 1997). Their specific method is a branch-and-bound approach relying heavily on dominance rules and cut sets, a kind of problem specific nogoods. They implicitly show the importance of nogoods to fathom the huge search space of RCPSP problems. Their method optimally solved all instances from J30 for the first time. Unfortunately, the number of cut sets grows exponentially in the number of activities, so that this method is only considered to be efficient for small problems.

Laborie (2005) developed a CSP-based method using ILOG SCHEDULER 6.1. Their method uses minimal conflict sets—set of activities causing a resource overload when running at the same time—as a branching scheme, and filtering algorithms including time-table and edge-finding. A conflict set is a set of activities whose execution might overlap in time and violate at least one resource constraint if they are executed at the same time. The conflict set is minimal if a removal of any activity of this set leads to the situation that all remaining activities can be concurrently run without violating any resource constraint. His method was the best published method so far on the J60, J90, and J120.

After publishing some parts of our results in Schutt *et al.* (2009) and publishing online the remaining parts on RCPSP, Horbach's work was published online (Horbach 2010). He proposed a hybrid approach that is similar to a lazy online SMT solver. The RCPSP is partially encoded as a SAT problem and lazily extended as the search progresses. If a so-called process variable—a Boolean variable reflecting that an activity is run at a particular time period (analogous to the variables B_{it} from Chap. 4 on page 82)—is set to *true* during unit propagation then the SAT solver is interrupted and a specialised integer linear solver checks for a resource overload at the corresponding time. In the case of a conflict, a conflict clause consisting of fixed process variables is returned to the SAT solver. If no overload occurs then the linear solver sets all unfixed processing variables which correspond to an activity that needs more resource units than left at this time period to *false*. As search he uses a VSIDS search (see Sec. 2.3 on page 23).

Horbach's approach uses the same idea as an LCG solver: using a SAT solver to benefit from the advanced SAT technology of its conflict learning facilities in order to prune the search space. In comparison to our generic approach proposed in this work his approach is hand-tailored for RCPSP. Moreover, his best results, which are comparable with ours, use the best known upper bound for the project duration

from the PSPLib as the initial upper bound on the planning horizon whereas our generic approach determines a first solution and takes its project duration as the initial upper bound on the planning horizon.

The best known upper bounds on the test sets J60, J90, and J120 were mostly computed by many highly specialised meta-heuristics. Therefore, it is believed that these upper bounds are either the optimum or a few time periods off the optimum.

5.2.2 RCPSP/max

RCPSP/max is a well studied problem with a number of challenging test sets from the PSPmax-library (Schwindt 2011): SM, CD, and UBO.[3] The first complete method we are aware of for it was proposed by Bartusch *et al.* (1988). They use a branch-and-bound algorithm to tackle the problem. Their branching is based on resolving (minimal) conflict sets—as in Laborie (2005)—by the addition of precedence constraints breaking these sets. Later other branch-and-bound methods were developed which are based on the same idea (see e.g. De Reyck and Herroelen 1998, Schwindt 1998a, Fest *et al.* 1999). The results from Schwindt are the best published ones for a complete method on the testset SM.

Dorndorf *et al.* (2000) use a time-oriented branch-and-bound combined with constraint propagation for precedence and resource constraints. In every branch one unscheduled and "eligible" activity is selected and its start time is assigned to the earliest period in time that does not violate any constraint regarding the current partial schedule. On backtracking they apply dominance rules to fathom the search space. As far as we can determine this complete approach outperforms other complete methods for RCPSP/max on the CD benchmark set.

Franck *et al.* (2001) compare different solution methods on the benchmark set UBO with instances having from 10 to 1000 activities. Their methods are truncated branch-and-bound algorithms, filter-beam search, heuristics with priority rules, genetic algorithms and tabu search. All methods share a preprocessing step to determine feasibility or infeasibility. The preprocessing step decomposes the precedence network into strongly connected components (SCCs) (which are denoted as "cyclic structures" in Franck *et al.* (2001)). The preprocessing then determines a solution or infeasibility for each SCC individually using constraint propagation techniques and a destructive lower bound computation where this computation first sets the planning horizon to a simple lower bound of the SCC and then increments the planning horizon until a feasible solution is found. Once a solution for all SCCs is determined a

[3] All test sets are accessible from `http://www.wior.uni-karlsruhe.de/LS_Neumann/Forschung/ProGenMax/` and via `http://129.187.106.231/psplib/`.

first solution can be deterministically generated for the original instance; otherwise infeasibility is proven.

Ballestín *et al.* (2009) employ an evolutionary algorithm based on a serial SGS with an unscheduling step. Their crossover operator is based on so called *conglomerates*, *i.e.*, sets of cycle structures and other activities which cannot move freely inside a schedule. It tries to keep the "good" conglomerates of the parents for their children. This is the best published meta-heuristic so far on the test sets UBO (instances with up to 100 activities) and CD.

Cesta *et al.* (2002) propose a two layered heuristic that is based on a temporal precedence network and an extension of this network by new temporal precedence relations in order to resolve minimal conflict sets. For guidance, constraint propagation techniques are applied on the network. Their method is competitive on the benchmark set SM.

Oddi and Rasconi (2009) apply a generic iterative search consisting of a relaxation and flattening step based on temporal precedence constraints which are used for resolving resource conflicts. In the first step some of the temporal precedence constraints are removed from the problem and then in the second step others are added if a resource conflict exists. Their method is evaluated on some larger instances from UBO.

In comparison to RCPSP not many meta-heuristics have been proposed due to the difficulty of handling generalised precedence relations. Moreover, the published results on test sets from PSPmax-library indicate a smaller gap between complete and incomplete methods than for RCPSP.

5.2.3 Other Related Works

Grimes *et al.* (2009), Grimes and Hebrard (2010) consider disjunctive resource scheduling problems, i.e., the resource capacity equals one for all resources. They apply their method to open-shop scheduling problems and job-shop scheduling problems with and without generalised precedence relations. These problems are restricted cases of RCPSP. An open-shop scheduling problem consists of a set of jobs which are made up by one non-preemptive activity for each resource. A feasible schedule orders the activities for each job and assigns a start time to each activity, so that no resource constraint is violated. A job-shop scheduling problem is an open-shop scheduling problem with a given order on the activities for each job, where a feasible schedule must obey generalised precedence relations if existent. As for RCPSP and RCPSP/max, a minimal schedule duration is sought. On both problem classes, their method is competitive and was able to close a number of open

problems.

Similar to our approach, they use nogood recording from restarts (Lecoutre *et al.* 2007) to avoid a repeated proving of infeasible parts of the search space in the subsequent search. Those nogoods are solely based on binary search decisions and are only generated before the search restarts. Consequently, propagators do not need to explain their propagation, and no nogoods are learnt between restarts. Compared to this approach, Lcg learns 1-Uip nogoods after each conflict. This allows Lcg not only to fathom the search space between restarts from nogoods, but also to build possibly stronger nogoods, since they are not only based on decisions. However, their approach seems to be well-suited to disjunctive scheduling, because they can exploit the binary relationship that one activity has to precede or follow another activity if they are from the same job or share the same resource. This is not the case for scheduling problems involving general cumulative constraints, since activities can be executed concurrently.

A number of linear programming methods have been proposed to solve Rcpsp (see, e.g., Brucker and Knust 2000, Baptiste and Demassey 2004, Damay *et al.* 2007). These methods relax the cumulative constraint to obtain a lower bound on the project duration. Several relaxations for the cumulative constraint can be found in these works and in Hooker and Yan (2002), Hooker (2007). Roughly, these methods work as follows: first, the problem is preprocessed, which can also involve extensive constraint propagation; second, the gathered information is used to generate an optimised mixed-integer programming (Mip) model using those relaxations. This in turn is solved with a Mip solver by sequential proving of infeasible lower bounds on the project duration until a feasible solution is found. These methods were successful in determining new lower bounds and closed several open problems. However, recently Vilím (2011) obtained comparable or better lower bounds by using a constraint programming solver with a hybrid of the time-table and edge-finding filtering algorithm for the cumulative constraint.

5.3 Resource-Constrained Project Scheduling Problems

Resource-constrained project scheduling problems (Rcpsp) appear as variants, extensions and restrictions in many real-world scheduling problems. Therefore, we test the *TimeD*, *ActiD* decompositions and the explaining cumulative propagator—presented in Chap. 4—on the well-known Rcpsp benchmark library PSPLib (Kolisch and Sprecher 1997). Before we evaluate our approaches and compare them with

state-of-the-art complete algorithms, we first present the model and different search strategies used to solve it.

5.3.1 Model

An RCPSP is denoted by a triple $(\mathcal{V}, \mathcal{E}, \mathcal{R})$ where $\mathcal{V}$ is a set of activities, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ a set of precedence relations between activities and $\mathcal{R}$ a set of resources. Each activity i has a processing time $p[i]$ and a resource usage $r[k, i]$ for each resource $k \in \mathcal{R}$. Each resource k has a resource capacity $R[k]$.

The goal is to find an optimal schedule that minimises the project duration MS, in the remainder of this section called *makespan*, while meeting the following conditions

$$\begin{aligned} \forall i \in \mathcal{V} : \quad & s[i] + p[i] \leq MS \ , \\ \forall i \ll j \in \mathcal{E} : \quad & s[i] + p[i] \leq s[j] \ , \\ \forall t \in [0\,..\,T-1], \forall k \in \mathcal{R} : \quad & \sum_{i \in \mathcal{V} : s[i] \leq t < s[i]+p[i]} r[k, i] \leq R[k] \ , \end{aligned}$$

where T is the planning horizon. This RCPSP problem can be written as a basic Zinc model.

```
1   % Parameters
2 int: T;                          % Planning horizon
3 enum Act;                        % Set of activities
4 enum Res;                        % Set of resources
5 set of int: Times = 0..T;        % Time periods
6 array[Act] of int: p;            % Processing times of activities
7 array[Act] of set of Act: suc;   % Successors of each activity
8 array[Res, Act] of int: r;       % Resource usages of activities
9 array[Res] of int: R;            % Resource capacities
10  % Variables
11 array[Act] of var Times: s;     % Start time variables of activities
12 var Times: MS;                  % Makespan of the project
13  % Precedence constraints
14 constraint
15   forall(i in Act, j in suc[i])( s[i] + p[i] <= s[j] );
16  % Resource constraints
17 constraint
18   forall(k in Res)(
19     cumulative(s, p, [i: r[k,i] | i in Act], R[k])
20   );
21  % Makespan constraints
22 constraint
23   forall(i in Act where suc[i] == {})( s[i] + p[i] <= MS );
```

```
24    % Objective
25 solve minimize MS;
```

A Zinc data file representing the problem of Ex. 4.1 on page 74 is

```
1 T = 20;
2 enum Act = { a, b, c, d, e, f };
3 enum Res = { single };
4 p = array1d(Act, [2,6,4,2,5,6]);
5 suc = array1d(Act, [{b},{c},{},{e},{},{}]);
6 r = array2d(Res, Act, [1,2,4,2,2,2]);
7 R = array1d(Res, [5]);
```

The basic model is refined with constraints expressing activities in disjunction, which is described next.

Activities in Disjunction

Two activities i and $j \in \mathcal{V}$ are in *disjunction*, if they cannot be executed at the same time, *i.e.*, the sum of their resource usages for at least one resource $k \in \mathcal{R}$ are greater than the available capacity: $r[i,k] + r[j,k] > R[k]$. Activities in disjunction can be exploited in order to reduce the search space.

The simplest way to model two activities i and j in disjunction is by two reified linear constraints sharing the same Boolean variable B_{ij}.

$$B_{ij} \rightarrow s[i] + p[i] \leq s[j] \qquad \neg B_{ij} \rightarrow s[j] + p[j] \leq s[i]$$

If B_{ij} is *true* then $i \ll j$, and if B_{ij} is *false* then $j \ll i$. The literals B_{ij} and $\neg B_{ij}$ can be directly represented in the SAT solver, consequently B_{ij} represents the relationship between these activities. The propagator of such a reified constraint can only infer new bounds on the left hand side of the implication if the right hand side is *false*, and on the start times variables if the left hand side is *true*. For example, the right hand side in the second constraint is *false* if and only if $\max_D s[i] - \min_D s[j] < p[j]$. In this case the literal $\neg B_{ij}$ must be *false* and therefore $i \ll j$.

We add these redundant constraints for all pairs of activities in disjunction to the model which allows the propagation solver to determine information about start time variables more quickly. The Zinc model of these constraints is as follows.

```
1  % Redundant non-overlapping (disjunctive) constraints
2 constraint
3   forall(i, j in Act where i < j)(
4     if exists(k in Res)(r[i, k] + r[j, k] > R[k]) then
```

```
5       % Activity i must be run before or after j
6       let {var bool: b} in (
7           (     b  -> s[i] + p[i] <= s[j])
8       /\  (not(b) -> s[j] + p[j] <= s[i])
9       )
10    else true endif
11  );
```

Further Remarks

In practice we share the Boolean variables generated inside the `cumulative` as described in Sec. 4.4.1 (by common sub-expression elimination) and add redundant constraints as described in Sec. 4.4.2 when using the *ActiD* decomposition. The planning horizon T is determined as the makespan of the first solution heuristically found by selection of the start time variable with the smallest lower bound (if a tie occurs then the lexicographic least variable) and assignment of the variable to its lower bound. The initial domain of each variable $s[i]$ was determined as $D_{init}(s[i]) = [\, head[i] \,..\, T - tail[i] \,]$ where $head[i]$ is the duration of the longest chain of predecessor activities, and $tail[i]$ is the duration of the longest chain of successor activities.

5.3.2 Search Strategies

We evaluate our approach on different search strategies. When we optimise an instance, the search strategies are combined with a branch-and-bound algorithm. Whenever a new solution is found (with $MS = t$), a constraint requiring a better solution ($MS < t$) is globally added during the search.

Search using Serial Scheduling Generation

Baptiste and Le Pape (2000) adapt the serial SGS (Kelley 1963)—as described in the related work section—for a constraint programming framework. For our experiments we use a form where we do not apply their dominance rules, and we impose a lower bound on the start time variable instead of posting the delaying constraint "activity i executes after at least one activity in the schedule".

1. *Select* an eligible unscheduled activity i with the earliest start time $t = lb(s[i])$. If there is a tie between some activities then select that one with the minimal latest start time $ub(s[i])$. If still tied then choose the lexicographic least activity. Create a choice point.

2. *Left branch:* Extend the partial schedule by setting $s[i] = t$. If this branch fails then go to the right branch. Otherwise go to step 1.

3. *Right branch:* Delay activity i by setting $s[i] \geq t'$ where $t' = \min\{lb(s[j])+p[j] \mid j \in T : lb(s[j]) + p[j] > lb(s[i])\}$, *i.e.*, the earliest completion time of the concurrent activities. If this branch fails then backtrack to the previous choice point. Otherwise go to step 1.

The right branch uses the dominance rule that amongst all optimal schedules there exists one where every activity starts either at the first possible time or immediately after the completion of another activity. Therefore, the imposing of the new lower bound is sound; no solution is lost for the considered problem. If we add side constraints then this assumption could be invalid. This search is simply referred as to Sgs.

Search using Variable State Independent Decaying Sum

As the Sat conflict driven search we use a variant of Vsids as described in Sec. 2.3 on page 23.

Restarting has been shown to be beneficial in Sat solving (and Csp solving) in speeding up solution finding, and as being more robust on hard problems. On restart the set of nogoods has changed as well as the activity of variables, so the search takes a different path. We also use Vsids search with restarting, which we denote as Restart. Note that restarting with Sgs is not very attractive since we rarely learn anything that changes the Sgs search decisions, we effectively just continue the same search.

Hybrid Search Strategies

One drawback of Vsids search is that at the beginning of the search the activity counters are only related to the clauses occurring in the original model, and not to any conflict. This is exacerbated in Lcg where many of the constraints of the problem may not appear at all in the clause database initially. This can lead to poor decisions in the early stages of the search. Our experiments support this: there are a number of "easy" instances which Sgs can solve within a small number of choice points, where Vsids requires substantially more.

In order to avoid these poor decisions we consider a hybrid search strategy. We use Sgs for the first 500 choice points and then restart the search with Vsids. The Sgs search may solve the whole problem if it is easy enough, but otherwise it sets the activity counters to meaningful values so that Vsids starts concentrating on

meaningful decisions. We denote this search as HOT START, and the version where the secondary VSIDS search also restarts as HOT RESTART.

5.3.3 Experiments

We carried out extensive experiments on RCPSP instances comparing our approaches to decomposition without explanation, global cumulative propagators from SICStus and ECLiPSe, as well as a state-of-the-art complete solving algorithms (Laborie 2005, Horbach 2010). Detailed results are available at `http://www.cs.mu.oz.au/~pjs/rcpsp`.

We use two suites of benchmarks. The library PSPLib contains the four classes J30, J60, J90, and J120 consisting of 480 instances of 30, 60, 90, and 120 activities respectively. We also use a suite (BL) of 40 highly cumulative instances with either 20 or 25 activities constructed by Baptiste and Le Pape (2000).

The experiments were run on a X86-64 architecture running GNU/Linux and a Intel(R) Xeon(R) CPU E54052 processor at 2GHz. The code was written in Mercury (Somogyi *et al.* 1996) using the G12 Constraint Programming Platform (Stuckey *et al.* 2005) and compiled with the Mercury Compiler and grade hlc.gc.trseg. Each run was given a 10 minute limit.

We compare 4 different implementations of the cumulative constraint with explanation: (`t`) the *TimeD* decomposition of Sec. 4.4.1, (`s`) the *ActiD* decomposition of Sec. 4.4.2, (`e`) an activity decomposition using end times, and (`g`) a global constraint `cumulative` using time-table filtering with explanation. The global `cumulative` uses the pointwise explanations for consistency and iterative pointwise explanations for filtering of Sec. 4.5. We experimented with other forms of explanations for the global `cumulative` but they were inferior for hard instances (about 15% worse on average except for naïve explanations which behave poorly). The present implementation of the global `cumulative` recalculates the resource profile on each invocation, which could be significantly improved by making it incremental. Profiling on a few instances showed that more than half of the time was spent in the propagation of `cumulative`.

Results on J30 and BL Instances

The first experiment compares different decompositions and search on the smallest instances J30 and BL. We compare SGS, VSIDS, RESTART, and the hybrid search approaches using our 4 different explanation approaches. The results are shown in Tab. 5.1 and 5.2. For J30 we show the number of problems solved (#`svd`),

Table 5.1: Results on J30 instances

			cmpr(477)		all(480)	
search	model	#svd	time	fails	time	fails
SGS	s	477	2.13	3069	5.86	5375
	e	477	2.19	3054	5.93	5331
	t	**480**	0.87	2339	2.83	4230
	g	**480**	0.73	1977	3.04	3919
VSIDS	s	**480**	1.20	2128	1.63	2984
	e	**480**	0.46	1504	0.77	2220
	t	**480**	0.26	1002	0.33	1271
	g	**480**	0.15	797	0.20	1058
RESTART	s	**480**	0.50	1483	0.93	2317
	e	**480**	0.43	1368	0.80	2128
	t	**480**	0.24	856	0.33	1174
	g	**480**	0.15	777	0.22	1093
HOT START	t	**480**	0.21	779	0.34	1220
	g	**480**	**0.12**	**706**	**0.17**	**956**
HOT RESTART	t	**480**	0.26	884	0.35	1231
	g	**480**	0.13	727	0.21	1058

(cmpr(477)) the average solving time in seconds and number of failures on the 477 problems that all approaches solved, and (all(480)) average solving time in seconds and number of failures on all 480 problems within the execution.[4] Note that we shall use similar comparisons and notation in future tables. For the BL problems the results are shown in Tab. 5.2. We show the number of solved problems, (all(40)) average solving time and number of failures with a 10 minute limit (on all 40 instances), as well as fails(4000) with a 4000 failure limit.

Of the decompositions the *TimeD* decomposition is clearly the best, being almost twice as fast as the *ActiD* decompositions. This is the effect of the stronger propagation. Note that these are the smallest problems where its relative disadvantage in size is least visible. The global is usually significantly better than the *TimeD* decomposition: it usually requires less search and can be up to twice as fast. Interestingly sometimes the *TimeD* decomposition is faster which may reflect the fact that it is (automatically) a completely incremental implementation of the cumulative constraint. For these small problems the best search strategy is HOT START since the overhead of restarting VSIDS does not pay off for these simple problems.

[4]This means that for problems that time out the number of failures is substantially larger than those which were solved before timeout.

Table 5.2: Results on BL instances

search	model	#svd	all(40)		#svd	fails(4000)	
SGS	s	**40**	2.51	9628	24	0.18	1261
	e	**40**	2.63	9443	24	0.15	1144
	t	**40**	0.82	5892	29	**0.04**	781
	g	**40**	0.88	5723	30	0.05	860
VSIDS	s	**40**	0.79	4436	31	0.16	1115
	e	**40**	0.77	4104	30	0.15	1025
	t	**40**	0.22	2540	34	**0.04**	661
	g	**40**	0.20	2039	**37**	**0.04**	605
RESTART	s	**40**	0.88	4549	31	0.17	1169
	e	**40**	1.46	5797	32	0.17	1135
	t	**40**	0.13	1626	35	0.05	603
	g	**40**	0.14	1568	36	**0.04**	**546**
HOT START	t	**40**	**0.10**	**1448**	36	**0.04**	680
	g	**40**	0.12	1485	36	**0.04**	593
HOT RESTART	t	**40**	0.15	1829	35	0.05	719
	g	**40**	0.25	2460	36	0.05	680

The results on the BL instances show that approaches using *TimeD* or the global propagator and VSIDS could solve between 6 and 9 instances more than the base approach (FE) of Baptiste and Le Pape (2000) within 4000 failures. Their "left-shift/right-shift" approach could solve all 40 instances in 30 minutes, with an average of 3634 failures and 39.4 seconds on a 200 MHz machine. All our approaches with *TimeD* and VSIDS find the optimal solution faster and in fewer failures (between a factor of 1.39 and 2.4).

Next we compare the *TimeD* decomposition (SGS+`t`) and global propagator (SGS+`g`) against implementations of `cumulative` in SICStus v4.0 (default, and with the flag `global`) and ECLiPSe v6.0 (using its 3 `cumulative` versions from the libraries `cumulative`, `edge_finder` and `edge_finder3`). We also compare against (FD+`t`) a decomposition without explanation (a normal FD solver) executed in the G12 system. All approaches use the SGS search strategy.

The results are shown in the Tab. 5.3 and 5.4. Clearly the more expensive edge-finding filtering algorithms are not advantageous on the J30 examples, but they do become significantly beneficial on the highly cumulative BL instances. We can see that none of the other approaches compare to the LCG approaches. The best solver without learning is the SICStus `cumulative` with `global` flag. Clearly nogoods are very important to fathom search space.

Table 5.3: Results of the FD solvers on the J30 instances

	solver	#svd	cmpr(364)		all(480)	
SICStus	default	418	0.22	337	87.43	141791
	global	415	0.40	331	94.01	76533
ECLiPSe	cumu	368	13.98	26469	154.79	365364
	ef	366	18.33	21717	157.43	173445
	ef3	368	16.61	17530	155.87	155142
G12	FD +t	404	1.79	5701	104.38	641185
	SGS +t	**480**	**0.01**	75	**2.83**	4230
	SGS +g	**480**	**0.01**	**70**	3.04	**3919**

Table 5.4: Results of the FD solvers on the BL instances

	solver	#svd	cmpr(7)		all(40)	
SICStus	default	32	2.87	25241	195.05	1896062
	global	39	0.90	3755	18.36	63310
ECLiPSe	cumu	7	178.90	352318	526.61	2231026
	ef	37	43.06	53545	102.60	229332
	ef3	37	35.56	39836	81.47	144051
G12	FD +t	30	6.71	72427	216.87	2650886
	SGS + t	40	**0.01**	**268**	**0.82**	5892
	SGS + g	40	**0.01**	278	0.88	**5723**

While the *TimeD* decomposition clearly outperforms *ActiD* on these small examples, as the planning horizon grows at some point *ActiD* should be better, since its model size is independent of the planning horizon. We took the J30 examples and multiplied the durations and planning horizon by 10 and 100. We compare the *TimeD* decomposition versus the (e) end-time *ActiD* decomposition (which is slightly better than start-time (s)) and the global `cumulative` (g). The results are shown in Tab. 5.5. First we should note that simply increasing the durations makes the problems significantly more difficult. While the *TimeD* decomposition is still just better than the *ActiD* decomposition for the 10× extended examples, it is inferior for scheduling problems with very long durations. The most important result visible from this experiment is the advantage of the global propagator over the *TimeD* decomposition as the planning horizon gets larger. The global propagator is by far the best approach for the larger problems since it has the $\mathcal{O}(n^2)$ complexity of the *ActiD* decomposition but the same propagation strength as the much stronger *TimeD* decomposition. Note also how the failures get dramatically worse for the

Table 5.5: Results on the modified J30 instances

search	duration	model	#svd	cmpr(462)		all(480)	
SGS	1×	e	477	0.11	542	5.93	5331
		t	**480**	0.08	501	2.83	4230
		g	**480**	**0.05**	**371**	**3.04**	**3919**
	10×	e	471	0.58	1812	14.76	9512
		t	476	1.03	676	11.04	4972
		g	**478**	**0.10**	**393**	**4.92**	**4291**
	100×	e	466	4.87	4586	23.40	10813
		t	465	15.58	724	35.02	**1582**
		g	**477**	**0.70**	**403**	**8.51**	3684
VSIDS	1×	e	**480**	0.06	318	0.77	2220
		t	**480**	0.04	249	0.33	1271
		g	**480**	**0.02**	**151**	**0.20**	**1058**
	10×	e	**480**	0.18	821	4.23	4213
		t	**480**	0.63	1210	4.84	2714
		g	**480**	**0.06**	**284**	**0.39**	**1215**
	100×	e	474	1.32	2031	12.36	4224
		t	469	9.88	9229	27.35	9707
		g	**480**	**0.62**	**1296**	**3.15**	**2360**

TimeD decomposition using VSIDS as the problem grows. This illustrates how the large number of Boolean variables in the decomposition makes the VSIDS heuristic less effective.

Results on J60, J90 and J120 Instances

We now examine the larger instances J60, J90 and J120 from PSPLib. For J60 we compare the most competitive search approaches from the previous subsection: VSIDS, RESTART, HOT START and HOT RESTART using the *TimeD* decomposition and global propagator. For this suite our solvers cannot solve all 480 instances within 10 minutes. The results are presented in Tab. 5.6. For these examples we show the average distance of the makespan from our best solution to the best known solution from PSPLib at the date of 23 April 2009 (most of which are generated by specialised heuristic methods), as well as the usual time and number of failures comparisons. Many of these are currently open problems. Our best approaches close 24 open instances (see the end of this section for details). While all of the methods are quite competitive we see that restarting is valuable for improving the average distance from the best known solution, and the hybrid approach HOT RESTART is

Table 5.6: Results on J60 instances for *TimeD* and global propagator

search	model	#svd	avg. dist.	cmpr(425)		all(480)	
VSIDS	t	426	4.4	4.85	7216	72.71	**41891**
	g	**430**	6.2	2.99	4943	67.13	52016
RESTART	t	428	4.5	3.53	5139	68.04	61558
	g	**430**	**3.7**	**2.50**	**4418**	**66.01**	60518
HOT START	t	429	9.3	2.91	4629	66.64	52812
	g	428	18.1	2.93	4848	66.19	57823
HOT RESTART	t	429	4.0	3.28	4982	66.60	60146
	g	**430**	3.9	2.60	4658	66.18	60652

Table 5.7: Results on J90 instances for *TimeD* and global propagator

search	model	#svd	avg. dist.	cmpr(395)		all(480)	
HOT RESTART	`t`	396	**7.5**	4.16	4364	108.90	90582
	`g`	**397**	**7.5**	**3.34**	**3950**	**108.57**	**90134**

Table 5.8: Results on J120 instances for *TimeD* and global propagator

search	model	#svd	avg. dist.	cmpr(274)		all(600)	
HOT RESTART	`t`	274	9.7	8.50	8543	329.46	**234897**
	`g`	**282**	**9.6**	**5.40**	**7103**	**324.51**	242343

marginally more robust than the others. Interestingly HOT START can clearly force the search into a less promising area than just plain VSIDS.

For the largest instances J90 and J120 we ran only HOT RESTART since it is the most robust search strategy, using the *TimeD* decomposition and the global propagator. The results are presented in the Tab. 5.7 and 5.8 which show that the global propagator is superior to the *TimeD* decomposition. In total we close 15 and 27 open instances in J90 and J120 respectively (see the end of this section for details).

We compare our best method HOT RESTART with either `t` or `g` to the methods by Laborie (2005) and Horbach (2010). Laborie's method was the best published method so far on the J60, J90, and J120 instances whereas Horbach's method was recently published—after the results of HOT RESTART + `t` were published in Schutt *et al.* (2009) and the results of HOT RESTART + `g` were published online at `http://www.cs.mu.oz.au/~pjs/rcpsp` and submitted for publication.

Tables 5.9, 5.10, and 5.11 show the percentage of optimally solved instances within a maximal solving time with the 2.0GHz processor speed of the machine on which our method was run. We roughly estimate that our machine is 1.5 times faster than

Table 5.9: Comparison between state-of-the-art methods on J60

	Hot Restart			
2.0 GHz	t	g	Horbach	Laborie
10s	84.8	**85.8**	83.1	-
200s	**89.2**	89.0	-	84.2
300s	-	-	88.1	-
600s	89.4	**89.6**	-	-
1200s	-	-	-	85.0
3600s	-	-	89.6	-

Table 5.10: Comparison between state-of-the-art methods on J90

	Hot Restart			
2.0 GHz	t	g	Horbach	Laborie
10s	79.8	**80.0**	79.0	-
200s	81.7	**81.9**	-	78.5
300s	-	-	81.5	-
600s	82.5	**82.7**	-	-
1200s	-	-	-	79.4
3600s	-	-	82.5	-

Laborie's machine taking into account the speeds of the processors: 2.0GHz vs. 1.4GHz, and that it has a similar speed as Horbach's machine: 2.0GHz vs. 2.2GHz. Clearly this comparison can only be seen as indicative. A dash "-" in the table means that no results were available with the corresponding time limit.

Our methods clearly outperform Laborie's method: for every class our methods were able to solve more problems within 10s than they could solve in half an hour on their machine. Interestingly, our solver could not solve six instances which were solved by others. We can also see that the advantage of the global propagator

Table 5.11: Comparison between state-of-the-art methods on J120

	Hot Restart			
2.0 GHz	t	g	Horbach	Laborie
10s	42.3	**42.7**	40.8	-
200s	45.2	**45.8**	-	41.3
300s	-	-	44.7	-
600s	45.7	**47.0**	-	-
1200s	-	-	-	41.7
1800s	-	-	46.0	-

increases with increasing problem size.

Our methods perform comparably to Horbach's method and find and prove faster optimal solutions. Considering that our methods are generic and start with a loose upper bound on the makespan in comparison to his hand-tailored method that starts with the best known upper bound on the makespan, it seems that our methods, especially Hot Restart + g, are superior. Note that the initial Sat model size of both methods also depends on the domain size of each start time variable. Hence, starting with a tight upper bound not only saves runtime to find a tight upper bound, but also decreases the model size.

Closed Instances

Table 5.12 lists all previously open instances (with respect to the PSPLib at the date of 29 April 2009, Laborie (2005), and Liess and Michelon (2008)) with their optimal makespan. We exclude the results of Horbach (2010), since they were published later than our results. These instances were closed by Hot Restart with the global `cumulative` or some other method:

(♣) Vsids + g,

(♠) Hot Restart + t,

(♡) lower bound computation by proof of the equality of lower and best known upper bound, and

(♢) Hot Restart and lower bound computation—Hot Restart decreased the previously best known upper bound to 95 and the lower bound computation proved the optimality of this new bound.

The optimal makespan of almost all closed instances corresponds to the previously best known upper bound found by meta-heuristics, except for the J120 instances 8_3 and 48_5 where our solver could reduce them by 1 to 95 and 110 respectively. Note that Hot Restart with the *TimeD* decomposition was also capable of closing 63 of these instances.

New Lower Bounds

Finally we used Hot Start + g to try to improve lower bounds of the remaining open problems, by searching for a solution to the problem with the makespan varying from the best known lower bound to the best known upper bound from PSPLib. We set the makespan to the best known lower bound and tried to find a solution, if this

Table 5.12: Closed instances

j60	Instance	5_10	9_2	9_4	14_1	14_10	17_8	21_9	25_1
	Makespan	81	82	87	61	72	85	89	114
	Instance	25_3	25_5♣	25_9	30_5	30_7	30_10	41_1	41_2
	Makespan	113	98	99	76	86	86	122	113
	Instance	41_6	41_9	46_4	46_5	46_6	46_7	46_9	46_10
	Makespan	134	131	74	91	90	78	69	88
j90	Instance	5_1	5_2	21_2♡	21_4	21_5	21_6	21_9	21_10
	Makespan	78	93	116	106	112	106	121	109
	Instance	26_5♠	37_1	37_4	37_5	37_8♡	37_9	37_10♡	42_2
	Makespan	85	110	123	126	119	123	123	102
	Instance	42_7	42_10						
	Makespan	87	90						
j120	Instance	1_3	1_8	1_10	2_2	8_3♢	21_2	21_7	22_3
	Makespan	125	109	108	75	95	117	111	96
	Instance	22_8	28_4	28_8	28_9	28_10	29_4	41_2	41_9
	Makespan	103	112	99	98	116	80	141	121
	Instance	42_5	42_8	48_1	48_5	48_8	48_9	48_10	49_3
	Makespan	120	113	100	110	116	113	111	96
	Instance	49_4	49_5	49_7	49_10	50_4♡			
	Makespan	96	89	99	97	100			

failed we increased the makespan by one and re-solved. If a solution was found it is optimal, if we can prove failure for a given makespan we have increased the lower bound. If the increased lower bound equaled the best known upper bound we have proved the optimality of the upper bound and closed the instance as well.

In total this method closed 6 more instances and improved the lower bound by 78 instances of the remaining 433 open instances. The improved lower bounds are listed in Tab. 5.13.

5.4 Resource-Constrained Project Scheduling with Generalised Precedence Relations

Resource-constrained project scheduling problems with generalised precedence relations (Rcpsp/max) are an important extension of Rcpsp. In this section we run our best approach for Rcpsp, *i.e.*, the global cumulative constraint with explanation, on the test sets from PSP/max-library (Schwindt 2011). First, we present a basic model and then two extensions of it. We then describe the search strategies used.

Table 5.13: New lower bounds on all instances

j60	Instance	9_3	9_5	9_6	9_8	9_9	9_10	25_2	25_4
	LB	99	80	105	94	98	88	95	105
	Instance	25_6	25_7	25_8	25_10	29_2	29_9	30_2	41_3
	LB	105	88	95	107	123	105	69	89
	Instance	41_5	41_10	45_3	45_4				
	LB	109	105	133	101				
j90	Instance	5_4	5_6	5_8	5_9	21_1	21_7	21_8	37_2
	LB	101	85	96	113	109	105	107	113
	Instance	37_6	41_3	41_7	46_4				
	LB	129	147	144	92				
j120	Instance	1_1	6_8	7_2	7_3	7_6	8_2	8_4	8_6
	LB	104	140	113	97	115	101	91	84
	Instance	9_4	26_2	26_4	26_5	26_7	26_8	26_9	26_10
	LB	84	158	160	138	144	167	160	177
	Instance	27_2	27_5	27_7	27_10	28_7	29_3	34_8	42_1
	LB	109	105	118	110	108	96	86	106
	Instance	46_1	46_2	46_3	46_5	46_7	46_9	46_10	47_1
	LB	171	186	162	135	155	156	174	129
	Instance	47_2	47_4	47_5	47_9	48_3	48_6	48_7	49_2
	LB	126	119	125	140	109	102	105	108
	Instance	53_3	53_4	53_9	54_7	54_10	60_2		
	LB	105	137	155	108	107	82		

5.4.1 Model

An RCPSP/max problem is similarly denoted as an RCPSP problem by a triple $(\mathcal{V}, \mathcal{E}, \mathcal{R})$ where $\mathcal{V}$ is a set of activities, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathbb{Z}$ a set of generalised precedence relations between activities and $\mathcal{R}$ a set of resources. Each activity i has a processing time $p[i]$ and a resource usage $r[k,i]$ for each resource $k \in \mathcal{R}$. Each resource k has a resource capacity $R[k]$.

The goal is to find a schedule that minimises the makespan MS of the project while satisfying all precedence and resource constraints:

$$\begin{aligned} \forall i \in \mathcal{V} : \quad & s[i] + p[i] \leq MS \ , \\ \forall (i,j,d_{ij}) \in \mathcal{E} : \quad & s[i] + d_{ij} \leq s[j] \ , \\ \forall t \in [\,0\,..\,T-1\,], \forall k \in \mathcal{R} : \quad & \sum_{i \in \mathcal{V} : s[i] \leq t < s[i]+p[i]} r[k,i] \leq R[k] \ , \end{aligned}$$

where T is the planning horizon.

Example 5.1. A simple example of an RCPSP/max problem consists of the five activities $[\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}]$ with start times $[s_\mathtt{a}, s_\mathtt{b}, s_\mathtt{c}, s_\mathtt{d}, s_\mathtt{e}]$, processing times $[2, 5, 3, 1, 2]$ and resource usages on a single resource $[3, 2, 1, 2, 2]$ with a resource capacity 4.

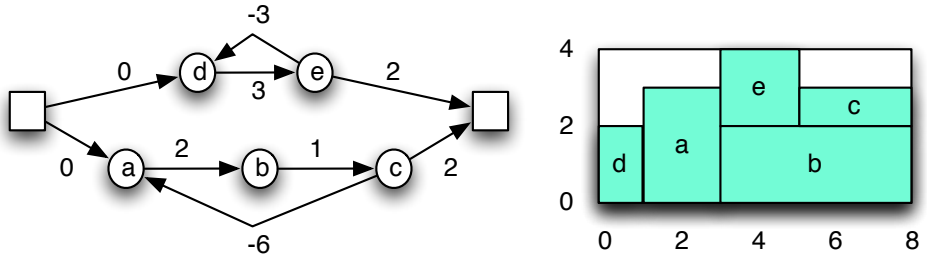


Figure 5.1: Left the activity-on-node network, and right a solution to a small RCPSP/max problem.

Suppose we also have the generalised precedences $s_{\mathtt{a}}+2 \leq s_{\mathtt{b}}$ (activity a ends before activity b starts), $s_{\mathtt{b}}+1 \leq s_{\mathtt{c}}$ (activity b starts at least 1 time period before activity c starts), $s_{\mathtt{c}}-6 \leq s_{\mathtt{a}}$ (activity c can not start later than 6 time periods after activity a starts), $s_{\mathtt{d}}+3 \leq s_{\mathtt{e}}$ (activity d starts at least 3 time periods before activity e starts), and $s_{\mathtt{e}}-3 \leq s_{\mathtt{d}}$ (activity e can not start later than 3 time periods after activity d starts). Note that the last two precedence constraints express the relation $s_{\mathtt{d}}+3 = s_{\mathtt{e}}$ (activity d starts exactly 3 time periods before activity e).

Let the planning horizon, in which all activities must be completed, be 8. Figure 5.1 illustrates the activity-on-node network between the five activities, the source at the left (time period 0), and sink at the right (time period 8), as well as a solution to this problem, where a rectangle for activity i has width equal to its duration and height equal to its resource usages. □

Generalised precedence relations $(i, j, d_{ij}) \in \mathcal{E}$ between the activities i and j are represented as the constraint $s[i] + d_{ij} \leq s[j]$, *i.e.*, it represents a *minimal time lag* (j must start at least d_{ij} time units after i starts) if $d_{ij} \geq 0$ and a *maximal time lag* (i must start at most $-d_{ij}$ time units after the start of j) if $d_{ij} < 0$. Generalised precedence relations encode not only start-to-start relations between activities, but also start-to-end, end-to-start, and end-to-end by addition/subtraction of i's or j's processing time to d_{ij}. If a minimal time lag d^+_{ij} and a maximal time lag d^-_{ji} exist for an activity j concerning to i then the start time $s[j]$ is restricted to $[s[i]+d^+_{ij}..s[i]-d^-_{ji}]$. In the case of $d^+_{ij} = -d^-_{ji}$ the activity j must start exactly d^+_{ij} time units after i.

A basic Zinc model for the RCPSP/max problem is illustrated as follows.

```
1  % Parameters
2  int: T;                          % Planning horizon
3  enum Act;                        % Set of activities
4  enum Res;                        % Set of resources
5  type prec = tuple(Act, Act, int); % (x, y, d) = x + d <= y
6  set of prec: E;                  % Set of generalised precedence relations
```

```
7  set of int: Times = 0..T;    % Time periods
8  array[Act] of int: p;        % Processing times of activities
9  array[Res, Act] of int: r;   % Resource usages of activities
10 array[Res] of int: R;        % Resource capacities
11   % Variables
12 array[Act] of var Times: s;   % Start time variables of activities
13 var Times: MS;                % Makespan of the project
14   % Generalised precedence constraints
15 constraint
16   forall(e in E)(s[e.1] + e.3 <= s[e.2]);
17   % Resource constraints
18 constraint
19   forall(k in Res)( cumulative(s, p, [i: r[k,i] | i in Act], R[k]) );
20   % Makespan constrains
21 constraint
22   forall(i in Act)(s[i] + p[i] <= MS);
23   % Objective
24 solve minimize MS;
```

A Zinc data file representing the problem of Ex. 5.1 is

```
1 T = 8;
2 enum Act = { a, b, c, d, e };
3 enum Res = { single };
4 p = array1d(Act, [2,5,3,1,2]);
5 E = { (a,b,2),(b,c,1),(c,a,-6),(d,e,3),(e,d,-3) };
6 r = array2d(Res, Act, [3,2,1,2,2]);
7 R = array1d(Res, [4]);
```

This basic model has a number of weaknesses: first, the initial domains of the start times are large; second, each precedence relation is modelled as an individual linear inequality propagator; and finally, the SAT solver in LCG has no structural information about activities in disjunction.

A smaller initial domain reduces the size of the problem because fewer Boolean variables are necessary to represent the integer domain in the SAT solver. It can be computed in a preprocess step by taking into account the precedence relations in $\mathcal{E}$ as described in the next paragraph. Individual propagators for precedence constraints may not be so bad for a small number of precedence relations, but for a larger number of propagators, their queuing behaviour may result in long and costly propagation sequences. A global propagator can efficiently adjust the time-bounds in $\mathcal{O}(n \log n + m)$ runtime (see Feydy *et al.* 2008). Reified precedence constraints can be used for modelling activities in disjunctions as described later in this section.

Initial Domain

A smaller initial domain can be obtained for the start time variables by applying the Bellman-Ford single source shortest path algorithm (see Bellman 1958, Ford and Fulkerson 1962)—or any other single shortest path algorithm that admits negative edge weights—on the digraph $G = (\mathcal{V}', \mathcal{E}')$ where $\mathcal{V}' = \mathcal{V} \cup \{v_0, v_{n+1}\}$,$\mathcal{E}' = \{(i, j, -d_{ij}) \mid (i, j, d_{ij}) \in \mathcal{E}\} \cup \{(v_0, i, 0), (i, v_{n+1}, -p_i) \mid i \in \mathcal{V}\}$, v_0 is the source node, and v_{n+1} is the sink node. The digraph is referred to as the activity-on-node network in the literature (e.g. Bartusch, Möhring, and Radermacher 1988, Neumann and Schwindt 1997). If the digraph contains a negative-weight cycle then the RCPSP/max instance is infeasible. Otherwise the shortest path from the source v_0 to an activity i determines the earliest possible start time for i, *i.e.*, $-w(v_0 \to i)$ where $w(.)$ is the length of the path and the shortest path from an activity i to the sink v_{n+1} the latest possible start time for i in any schedule, *i.e.*, $t_{max} + w(i \to v_{n+1})$. The Bellman-Ford algorithm has a runtime complexity of $\mathcal{O}(|\mathcal{V}| \times |\mathcal{E}|)$.

These earliest and latest start times can not only be used for initial smaller domains, but also to improve the objective constraints by replacing them with

```
1 constraint
2   forall(i in Act)(s[i] + tail[i] <= MS);
```

where `tail[i]` is the "negative" length $-w(i \to v_{n+1})$ of the shortest path from i to v_{n+1} in the digraph G. Preliminary experiments confirmed that starting the solution process with a smaller initial domain offers major improvements for solving an instance and generating a first solution, especially on larger instances. Another specific advantage for LCG is that a smaller initial domain also reduces the size of the problem because fewer Boolean variables are necessary to represent the integer domain in the SAT solver.

Activities in Disjunction

Activities in disjunction for RCPSP/max are the same as for RCPSP (see Subsec. 5.3.1). Two activities i and $j \in \mathcal{V}$ are in *disjunction*, if they cannot be executed at the same time, *i.e.*, their resource usages for at least one resource $k \in \mathcal{R}$ are bigger than the available capacity: $r[i,k] + r[j,k] > R[k]$. We extend our model with two reified constraints in the same way as for RCPSP.

5.4.2 Search Strategies

We use branch-and-bound algorithms based on deterministic and conflict driven branching strategies similar to RCPSP. Either the branching strategies are used

alone or in combination. After each branch all constraints are propagated until a fixpoint is reached or the inconsistency for the partial schedule is detected. In the first case a new node is explored and in the second case an unexplored branch is chosen if one exists or backtracking is performed.

Search using Deterministic Branching

The deterministic branching strategy selects an unfixed start time variable $s[i]$ with the smallest possible start time $\min_D s[i]$. If there is a tie between several variables then the variable with the biggest size, *i.e.*, $\max_D s[i] - \min_D s[i]$, is chosen. If there is still a tie then the variable with the lowest index i is selected. The binary branching is as follows: left branch $s[i] \leq \min_D s[i]$, and right branch $s[i] > \min_D s[i]$. In the remainder this branching is denoted by MSLF.

This branching creates a time-oriented branch-and-bound algorithm similar to Dorndorf *et al.* (2000), but it is simpler and does not involves any dominance rules. Hence, it is weaker than their algorithm.

Search using Conflict-driven Branching

We use the same conflict-driven search, VSIDS, as for RCPSP problems. In order to accelerate the finding of solutions and increase the robustness of the search on hard instances, VSIDS can be combined with restarts. In the remainder VSIDS with restart is denoted by RESTART. Different restart policies can be applied. Here, a geometric restart on failed nodes with an initial limit of 250 and a restart factor of 2.0 is used.

Hybrid Branching

As for RCPSP, the activity counters of the variables have to be initialised somehow at the beginning of the search. By default they are all initialised to the same value. With no useful information in this initial setting, these activities can mislead VSIDS resulting in poor performance. To avoid this, we consider a hybrid search that uses MSLF to search initially, which has the effect of modifying the activity counts to reflect some structure of the problem, and then switch to VSIDS after the first restart. Here, we switch the searches after exploration of the first 500 nodes unless otherwise stated. The strategy is denoted by HOT START, and HOT RESTART where VSIDS is combined with restart.

5.4.3 Experiments

We carried out experiments on RCPSP/max instances available at `http://www.wior.uni-karlsruhe.de/LS_Neumann/Forschung/ProGenMax/rcpspmax.html`.

We compare our LCG approaches using the explaining global cumulative propagator to the best known complete and incomplete methods so far on each testset. At the website `http://www.cs.mu.oz.au/~pjs/rcpsp` detailed results can be obtained.

Our methods are evaluated on the following test sets which were systematically created using the instance generator ProGen/max (Schwindt 1995):

CD: C, and D: each consisting of 540 instances with 100 activities and 5 resources.

UBO: UBO10, UBO20, UBO50, UBO100, and UBO200: each containing 90 instances with 5 resources and 10, 20, 50, 100, and 200 activities respectively. (cf. Franck *et al.* 2001).

SM: J10, J20, and J30: each containing 270 instances with 5 resources and 10, 20, and 30 activities respectively. (cf. Kolisch *et al.* 1998).

Note that although the testset SM consists of small instances they are considerably harder than e.g. UBO10 and UBO20.

The experiments were run on an Intel(R) Xeon(R) CPU E54052 processor with 2 GHz clock running GNU/Linux. The code was written in Mercury (Somogyi *et al.* 1996) using the G12 Constraint Programming Platform (Stuckey *et al.* 2005) and compiled with the Mercury Compiler using grade hlc.gc.trseg. Each run was given a 10 minute runtime limit.

Setup and Table Notations

In order to solve each instance a two-phase process was used. Both phases used the basic model with the two described extensions (cf. Subsection 5.4.1).

In the first phase a HOT START search was run to determine a first solution or to prove the infeasibility of the instance. The feasibility runs were set up with the trivial upper bound on the makespan $T = \sum_{i \in \mathcal{V}} \max(p[i], \max\{d_{ij} \mid (i, j, d_{ij}) \in \mathcal{E}\})$. The feasibility test was run until a solution was found or infeasibility proved. If a solution was found we use *UB* to denote the makespan of the resulting solution. In the first phase the search strategy should be good at both finding a solution or proving infeasibility, but not necessarily at finding and proving the optimal solution. Hence, it could be exchanged with methods that might be more suitable than HOT START. In contrast to the normal HOT START in the second phase we give the

deterministic search in the first phase more time to find a first solution and therefore we switch the strategies after $5 \times n$ nodes were explored where n is the number of activities.

In the second optimisation phase, each feasible instance was set up again this time with $T = UB$. The tighter bound is highly beneficial to LCG since it reduces the number of Boolean variables required to represent the problem. The search for optimality was performed using one of the various search strategies defined in the previous subsection.

The execution of the two-phased process leads to the following measurements.

rt_{max}**:** The runtime limit in seconds (for both phases together).

rt_{avg}**:** The average runtime in seconds (for both phases).

fails: The average number of infeasible nodes encountered in both phases of the search.

feas: The percentage of instances for which a solution was found.

infeas: The percentage of instances for which the infeasibility was proven.

opt: The percentage of instances for which an optimal solution was found and proven.

Δ_{LB}**:** The average distance from the best known lower bounds of feasible instances given in (Schwindt 2011).

cmpr(i): Columns with this header give measurements only related to those instances that were optimally solved by each procedure where i is the number of these instances.

all(i): Columns with this header compare measurements for all instances examined in the experiment where i is the number of these instances.

Entries with the symbol "-" indicate no comparable number was available. Entries with two numbers indicate that the corresponding procedure was applied several times to the instance and the first number gives the average over all runs and the number in parentheses gives the best number from all runs. Entries marked by a star "⋆" indicate that a procedure was not able to find a solution for all feasible instances and therefore the corresponding number may not be comparable with the number from other procedures.

Table 5.14: Comparison on the test sets CD, UBO, and SM.

					cmpr(2230)		all(2340)	
Procedure	feas	opt	infeas	Δ_{LB}	rt_{avg}	fails	rt_{avg}	fails
MSLF	85.0	80.60	15.0	3.96785	7.73	6804	35.96	23781
MSLF + restart	85.0	80.60	15.0	3.96352	7.80	6793	36.04	23787
VSIDS	85.0	82.26	15.0	3.76928	2.16	1567	22.91	13211
RESTART	85.0	82.26	15.0	3.73334	**2.02**	**1363**	22.38	**12212**
HOT START	85.0	82.31	15.0	3.84003	2.22	1684	22.71	12933
HOT RESTART	85.0	**82.35**	15.0	**3.73049**	2.04	1475	**22.36**	12341

Comparison of the Different Strategies

In the first experiment we compare all of our search strategies against each other on all test sets. The strategies are compared in terms of rt_{avg} and failures for each testset.

The results are summarised in the Tab. 5.14. Similar to the results for RCPSP all strategies using VSIDS are superior to the deterministic methods (MSLF), and similarly competitive. HOT RESTART is the most robust strategy, solving the most instances to optimality and having the lowest Δ_{LB}. Restart makes the search more robust for the conflict-driven strategies, whereas the impact of restart on MSLF is minimal.

In contrast to the results for RCPSP the conflict-driven searches were not uniformly superior to MSLF. The three instances 67, 68, and 154 from J30 were solved to optimality by MSLF and MSLF with restart, but neither RESTART and HOT RESTART could prove the optimality in the given time limit, whereas VSIDS and HOT START were not even able to find an optimal solution within the time limit. Furthermore, our method could not find a first solution for the UBO200 instances 2, 4, and 70 nor prove the infeasibility for the UBO200 instance 40 within 10 minutes. Only for these instances we let our method run until a first solution was found or infeasibility was proven. The corresponding numbers are included in Tab. 5.14. A detailed discussion of these instances follows later in this subsection.

Results on CD instances

Table 5.15 presents the results for the testset CD where 98.1% (1.9%) of the instances are feasible (infeasible). Here, we compare RESTART and HOT RESTART with the time-oriented branch-and-bound procedure ($B\&B_{D00}$) from Dorndorf *et al.* (2000) and the evolutionary algorithm EVA from Ballestín *et al.* (2009). The method

Table 5.15: Results on the testset CD.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}
B&B$_{\text{D00}}$	100	-	**98.1**	71.7	**1.9**	4.6▲
Eva	-	0.62	98.1	$\geq$ 65.9	-	**3.24 (3.16)**
Restart	1	0.38	97.9	**78.1**	1.6	4.73⋆
	10	1.39	98.1	**89.8**	1.9	**3.20**
	100	6.17	98.1	**94.0**	1.9	**2.86**
	600	19.32	98.1	95.8	1.9	2.81
Hot Restart	1	0.44	97.9	76.8	1.6	4.87⋆
	10	1.49	98.1	89.6	1.9	**3.20**
	100	6.27	98.1	93.9	1.9	**2.86**
	600	19.42	98.1	**96.0**	1.9	**2.79**

▲ The Δ_{LB} entry is based on the lower bounds presented in Schwindt (1998b) which were not accessible for us.

B&B$_{\text{D00}}$ performs better on this testset than the methods proposed by De Reyck and Herroelen (1998), Schwindt (1998a), Fest *et al.* (1999).[5] Moreover, B&B$_{\text{D00}}$ is the best published complete method on this testset so far. Their B&B$_{\text{D00}}$ method was implemented in C++ using Ilog Solver and Ilog Scheduler. Their experiments were run on a Pentium Pro/200 PC, thus their results were obtained on a machine approximately ten times slower.

We compare our results achieved with a runtime limit of 1 second to their results with a limit of 100 seconds which should be clearly in favour of them. While B&B$_{\text{D00}}$ can prove feasibility and infeasibility of all instances, the first-phase Hot Start search with one second was unable to prove infeasibility of four infeasible instances or find solutions to two feasible instances. It does prove infeasibility of these four infeasible instances in less than 2.1 seconds and finds a first solution for these two feasible instances in 4.8 seconds and 5.04 seconds respectively. Within one second both our methods Restart and Hot Restart were able to prove the optimality of substantially more instances than B&B$_{\text{D00}}$. With more time our methods are able to prove optimality of almost all instances in the test sets.

One reason for the first-phase results at one second may simply be that there is a reasonable set up time required for Lcg to generate all the Boolean variables and hence there is not much time for search. Another reason for the weakness of proving infeasibility is that our model only contains propagators that determine the order of activities in disjunction concerning their domains, but not also their minimal

[5]Results from Schwindt (1998a) are taken from Dorndorf *et al.* (2000).

distance in the transitive closure of all precedences.[6] Dorndorf *et al.* (2000) show that these propagators are important for a fast detection of infeasibility. That HOT START is not so good in finding a first solution is not surprising, since the search is not as problem specific as that of B&B$_{\text{D00}}$. In order to overcome these problems we could replace our first phase with e.g. the method of B&B$_{\text{D00}}$ to prove infeasibility and generate a first solution, and then use our second phase approach to find and prove optimality.

The method EVA is the best published meta-heuristic on this testset. Their results were obtained on a Samsung X15 Plus computer with Pentium M processor with 1400 MHz clock speed. This means that our machine is at least 1.46 times faster than theirs. Their limits are a maximum of 5000 schedules, halting the process at any time after 10 generations where the best schedule could not be improved. Our methods generate better schedules within 10 seconds than their approach, which can be seen from the lower Δ_{LB} 3.20 which is less than 3.24.

Overall our methods are able to close 310 open problems and improve the upper bound for all 21 remaining open problems in testset CD, according to the results recorded in Schwindt (2011).

Results on UBO instances

Table 5.16 compares our procedures RESTART and HOT RESTART with the truncated branch-and-bound methods FBS$_{\text{F01}}$, the heuristic DM$_{\text{F01}}$, and the genetic algorithm GA$_{\text{F01}}$ all proposed by Franck *et al.* (2001) on the UBO testset where 81.7% (18.3%) of the instances are feasible (infeasible). In this table we add the column feas + infeas showing the sum of percentage of feas and infeas because the corresponding numbers for FBS$_{\text{F01}}$ are not available. Their results were obtained on a Pentium II machine with a 333MHz processor, *i.e.*, our machine is at least 6.2 times faster. They imposed a time limit of n seconds, e.g. an instance with 100 activities was given at most 100 seconds. We compare our methods with 10 or 100 times lower time limit which should be favourable to the other methods.

Their methods were able to prove the feasibility or infeasibility for all instances (except one instance for the method FBS$_{\text{F01}}$). Indeed DM$_{\text{F01}}$ is extremely fast requiring just 0.03 seconds on average, but it does not necessarily find very good solutions, as shown by the high Δ_{LB}.

In contrast our first-phase was not always able to find a first solution or prove infeasibility with the time limit $n/100$. No solution was found for 6 instances with 100 activities, and the infeasibility was not shown for 11 (1) instances with 100 (50)

[6] These propagators are not available in the G12 Constraint Programming Platform.

Table 5.16: Results on the testset UBO for UBO10, UBO20, UBO50 and UBO100 instances in comparison with FBS_{F01}, DM_{F01}, and GA_{F01}.

Procedure	rt_{max}	rt_{avg}	feas + infeas	feas	opt	infeas	Δ_{LB}
FBS_{F01}	n	12.4	99.66	-	-	-	6.82*
DM_{F01}	n	0.03	100	81.7	-	18.3	10.72
GA_{F01}	n	3.16	100	81.7	-	18.3	6.93
RESTART	$n/100$	0.21	95.0	80.0	70.8	15.0	5.73*
	$n/10$	0.78	100	81.7	75.3	18.3	4.99
HOT RESTART	$n/100$	0.25	95.0	80.0	69.7	15.0	5.73*
	$n/10$	0.81	100	81.7	75.3	18.3	5.04

Table 5.17: Results on the testset UBO for UBO10, UBO20, UBO50 and UBO100 instances in comparison with EVA.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}
EVA	-	0.38	81.7	-	-	**4.82 (4.79)**
RESTART	1	0.22	80.0	**71.4**	15.3	5.60*
	10	0.89	81.7	**75.3**	18.3	4.92
	100	5.32	81.7	**77.2**	18.3	**4.51**
	600	24.47	81.7	**78.1**	18.3	**4.40**
HOT RESTART	1	0.26	80.0	70.6	15.3	5.65*
	10	0.92	81.7	**75.3**	18.3	5.01
	100	5.26	81.7	**77.2**	18.3	4.55
	600	24.14	81.7	**78.1**	18.3	4.43

activities. Once the time limit was extended to $n/10$ then the first phase was always able to find a solution or prove infeasibility. If we compare Δ_{LB} achieved with a time limit $n/10$ (note for a time limit $n/100$ the data is not comparable, since our methods could not find a solution for all feasible instances) then our methods have a substantially better Δ_{LB} than their approaches, *i.e.*, our methods are quicker in improving the makespan. Our approaches could prove optimality for a substantial fraction of these problems even with time limit $n/100$.

Table 5.17 compares our results with the best meta-heuristic EVA from Ballestín *et al.* (2009) on the UBO instances with up to 100 activities. Our methods create better schedules within 100 seconds than the evolutionary algorithm EVA, leading to a smaller lower bound deviation.

Table 5.18 presents the results on UBO200 which are compared to the iterative flattening searches IFS, IFS-FR, and IFS-MCSR from Oddi and Rasconi (2009).[7]

[7]No machine details are given in Oddi and Rasconi (2009).

Table 5.18: Results on the testset UBO for UBO200 instances.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}	Δ_{UB}
IFS	-	2148.7	88.9	-	-	-	2.06
IFS-FR	-	2024.7	88.9	-	-	-	1.81
IFS-MCSR	-	1716.7	88.9	-	-	-	1.65
RESTART	100	29.55	81.1	67.8	7.8	7.37*	-0.41*
	600	139.0	85.6	**68.9**	10.0	10.11*	-1.110*
	600+$^\blacktriangle$	187.5	88.9	**68.9**	11.1	11.88	-1.249
HOT RESTART	100	29.9	81.1	**68.9**	7.8	**7.22***	**-0.48***
	600	139.0	85.6	**68.9**	10.0	**10.10***	**-1.111***
	600+$^\blacktriangle$	186.9	88.9	**68.9**	11.1	**11.87**	**-1.250**

$^\blacktriangle$ For comparison purpose the instances 2, 4, 40, and 70 were run until a first solution was found or infeasibility proven.

The table contains the extra column Δ_{UB} that reports the average distance from the best known upper bounds of feasible instances given in Schwindt (2011). Here, 88.9% (11.1%) instances are feasible (infeasible). Note that Franck *et al.* (2001) also run their methods on UBO200, but the presented results are accumulated with the results on instances with 500 and 1000 activities, so that a comparison is not possible.

Within the given time limit HOT START was not able to find a solution for the instances 2, 4, and 70 nor to prove the infeasibility for the instance 40. In order to compare the results with Oddi and Rasconi (2009) we let our first solution method run until a solution was found or infeasibility proven. The runtimes for the instances 2, 4, 40, and 70 are 1030, 1478, 1139, and 3103 seconds, respectively. Interestingly, the first obtained solutions have a better upper bound by 62, 46, and 37 time periods for the instances 2, 4, and 70 respectively than the previously best known upper bound recorded in Schwindt (2011).

Comparing these results on Δ_{UB} with Oddi and Rasconi (2009) clearly our procedures achieve better schedules. RESTART and HOT RESTART perform comparably. The UBO200 instances clearly show that HOT START as the search strategy in the first phase can have difficulties in finding a first solution or proving infeasibility.

In total our approaches close 178 open instances and improve the upper bound for 27 instances of 31 remaining open instances with 200 activities or less in the testset UBO, according to the results recorded in Schwindt (2011).

Results on SM instances

Finally for the testset SM we compare our approaches MSLF, RESTART, and HOT RESTART with the method B&B$_{S98}$ from Schwindt (1998a)[8], ISES from Cesta *et al.* (2002), and SWO(BR) from Smith and Pyle (2004). The method B&B$_{S98}$ (Schwindt 1998a) is a branch-and-bound algorithm that resolves resource conflicts by adding precedence constraints between activities and has been run on a Pentium 200 with a 100 second time limit. ISES is a heuristic that also adds precedence constraints between activities in order to resolve/avoid resource conflicts, uses restarts and has been run on a SUN UltraSparc 30 (266 MHz) with the same time limit. The method SWO(BR) (Smith and Pyle 2004) is a squeaky wheel optimisation. Their method is divided into two stages: schedule generation and prioritisation where the schedule is created by a heuristic with priority scheme and the latter changes the priorities on variables depending on how "well" it is handled in the former stage. Their benchmarks were performed on a 1700 Mhz Pentium 4. Note that ISES and SWO(BR) are incomplete methods, *i.e.*, they cannot prove infeasibility unless the precedence graph contains a positive weight cycle, and optimality is only proven if the makespan of the solution found equals the known lower bound.

Table 5.19 presents the results for the 270 instances from SM with 30 activities. From these instances 185, *i.e.*, 68.5%, are feasible and 85, *i.e.*, 31.5%, infeasible. All our approaches could prove feasibility and infeasibility of all instances within one second whereas B&B$_{S98}$ could not find a solution for a few feasible instances. Moreover, our methods could prove optimality significantly more often than the complete method B&B$_{S98}$ (of course also the incomplete methods). All our methods were able to find on average a better solution in one second than these approaches, as indicated by a lower Δ_{LB}. For these harder benchmarks our methods clearly outperform the competition. One reason could be that constraint propagation over the cumulative constraint has a greater benefit than on other test sets because here more activities can be run simultaneously.

Our approaches each give similar results: RESTART and HOT RESTART are superior to MSLF up to 10 seconds, and all are similar to each other with longer time limits. On the one hand MSLF could prove optimality for three instances where RESTART and HOT RESTART only found the optimal solution. On the other hand MSLF could not find an optimal solution for two instances where RESTART and HOT RESTART could. It seems that MSLF may better suit problems where more activities can be executed in parallel, but this needs further investigation.

[8]The paper (Schwindt 1998a) was not accessible to us, so that here the reported results are taken from Cesta *et al.* (2002).

Table 5.19: Results on the J30.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}
B&B$_{S98}$	100	-	67.7	42.6	-	9.56▲
ISES	100	22.68	68.5	33.9 (35.6)	-	10.99 (10.37)
SWO(BR)	10	1.07	68.5	35.0	-	10.3
MSLF	1	0.16	68.5	58.1	31.5	8.91
	10	0.82	68.5	61.9	31.5	8.40
	100	4.90	68.5	**64.8**	31.5	8.23
	600	21.61	68.5	**65.5**	31.5	8.20
RESTART	1	0.12	68.5	**61.5**	31.5	8.38
	10	0.57	68.5	**64.1**	31.5	8.19
	100	3.92	68.5	**64.8**	31.5	8.17
	600	21.34	68.5	65.2	31.5	**8.12**
HOT RESTART	1	0.12	68.5	**61.5**	31.5	**8.37**
	10	0.59	68.5	**64.4**	31.5	**8.18**
	100	3.93	68.5	**64.8**	31.5	**8.16**
	600	21.47	68.5	65.2	31.5	8.13

The Δ_{LB} entry marked by ▲ is based on the lower bounds presented in Schwindt (1998b) which were not accessible for us.

Experiments were also carried out on the instances with 10 or 20 activities. All our methods could solve all 270 instances with 10 activities within 0.05 seconds each. Also, all our methods could solve all 270 instances with 20 activities within 30 seconds each. Moreover for the instances with 20 activities an optimal solution was found within 1 second for all feasible instances.

Our approaches close 85 open problems and improve the upper bound for 3 problems of the 6 remaining open problems in the test set SM, according to the results recorded in Schwindt (2011).

In the remainder of this section we list all closed instances and those instances for which we improve the upper bound on the makespan.

Closed Instances

Tables 5.20–5.28 list all 573 previously open instances closed by one of our methods. For each instance the following parameters are given: the instance number (Inst), the previously best known upper bound (Best UB) on the makespan, the proved optimal makespan (Optimal) and the lowest runtime (Best rt) of our methods which could solve the instance to optimality. Optimal makespans are written in italics if the makespan is lower than the previously best known upper bound.

Table 5.20: All closed instances from class C.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
1	336	336	0.61	3	379	379	0.24	4	258	258	0.70	6	336	*327*	0.44
12	331	331	0.24	32	370	*367*	6.12	33	383	383	2.43	34	421	*391*	60.40
35	259	*254*	221.81	37	325	*312*	129.92	38	306	*291*	40.96	39	428	*421*	5.10
40	395	*386*	1.45	62	621	*602*	10.72	64	688	688	0.39	65	376	*355*	60.37
90	293	293	0.19	91	260	260	0.74	92	360	360	0.24	94	428	428	0.22
95	399	399	0.58	96	501	501	0.81	97	489	489	0.53	98	518	518	0.55
100	399	399	0.42	121	410	*399*	6.19	124	304	*270*	26.90	126	502	502	2.64
127	404	*401*	0.24	128	505	505	0.53	129	517	*506*	9.26	130	434	*417*	28.87
153	554	*553*	3.76	154	535	*524*	12.41	155	375	*361*	201.56	156	399	*387*	134.98
157	475	*453*	1.89	158	397	397	1.28	159	488	488	2.44	165	371	371	3.17
181	456	456	0.36	182	376	376	0.37	183	461	461	0.45	185	370	*364*	2.36
186	410	410	0.40	188	321	*307*	0.97	190	401	401	0.19	191	493	493	0.13
211	445	445	0.88	212	564	564	0.89	213	710	710	1.01	214	624	624	1.56
217	365	*362*	2.64	220	403	*393*	1.95	224	304	304	0.45	242	431	*425*	76.30
243	533	*519*	7.76	244	514	*508*	4.74	246	574	574	1.54	247	478	*471*	15.98
248	443	*430*	31.29	249	635	*633*	1.95	251	308	308	1.21	260	469	469	0.54
271	498	*497*	2.84	272	277	277	0.39	273	598	*579*	1.05	277	448	*410*	2.08
278	587	587	0.27	280	451	451	0.30	301	412	412	1.33	303	329	*319*	1.53
304	346	*333*	132.18	306	296	*288*	1.00	307	342	*309*	110.16	308	564	*545*	19.74
309	503	503	2.87	314	329	*324*	2.03	315	294	294	0.49	328	255	255	0.34
332	336	*326*	15.10	333	410	*404*	3.71	335	426	*413*	3.03	338	415	*403*	145.06
340	322	*312*	12.46	346	446	446	8.43	349	451	*444*	17.53	361	523	*513*	5.18
363	566	566	1.77	364	372	*360*	0.36	365	445	445	0.65	366	419	419	0.25
367	322	322	0.32	369	390	390	0.21	391	323	*314*	6.78	392	322	*311*	30.72
393	337	*331*	0.37	394	469	469	0.50	397	588	*524*	8.76	399	315	*290*	72.91
400	420	*411*	2.21	406	362	362	0.22	413	344	344	0.36	421	469	*458*	7.04
422	794	*776*	33.42	423	401	401	1.11	424	394	*382*	74.41	426	350	*333*	302.85
427	314	*308*	2.07	428	831	831	30.62	430	361	*345*	108.07	433	372	*369*	9.68
435	361	*359*	3.53	440	260	*258*	5.52	451	365	365	0.25	452	420	*419*	0.62
453	659	659	5.45	454	498	*493*	0.65	455	304	304	0.29	456	609	609	0.26
457	430	*428*	0.29	458	402	402	0.24	459	499	*447*	1.26	481	433	*420*	1.22
482	905	905	16.12	483	426	*402*	4.73	484	574	574	0.58	486	586	*568*	15.16
487	734	734	11.47	488	485	*483*	0.46	489	397	*382*	6.55	490	462	462	0.22
493	503	503	0.21	495	353	353	0.16	497	333	*323*	2.58	511	440	440	1.32
513	555	*551*	0.80	514	501	*489*	3.21	515	715	*673*	19.99	516	394	*393*	1.17
517	407	*399*	0.78	518	424	*418*	8.86	519	437	437	0.78	520	567	*560*	1.38
523	389	389	1.86	530	292	292	0.17	538	308	308	0.38	540	310	310	8.64

Table 5.21: All closed instances from class D.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
2	488	488	0.71	3	359	*351*	1.22	6	483	*475*	1.90	7	371	371	0.47
9	558	*552*	7.16	10	430	*428*	1.89	11	400	400	0.13	31	581	*562*	35.64
32	603	*588*	285.56	33	448	*445*	46.95	34	489	*466*	5.93	36	674	674	11.08
37	529	529	0.67	38	496	*490*	2.00	40	491	491	0.80	61	476	476	2.02
62	717	*710*	9.69	64	611	*596*	6.12	65	539	*493*	138.61	66	472	*449*	323.21
67	501	*483*	4.92	68	582	*554*	123.64	70	622	*613*	235.48	71	356	*354*	2.34
88	394	394	1.12	91	502	502	0.41	92	407	407	0.28	93	392	*387*	1.09
94	457	457	1.13	96	450	*445*	0.22	97	468	*464*	2.49	98	540	*527*	0.85
100	447	447	0.49	122	665	*656*	4.19	123	497	*481*	4.63	124	634	*623*	7.57
125	492	*491*	0.76	126	413	*393*	40.83	127	445	*432*	0.95	128	661	*626*	43.29
130	623	*616*	13.47	134	455	*454*	0.76	151	575	*560*	7.24	152	467	*456*	1.25
153	535	*514*	204.89	155	463	*440*	122.23	156	545	*514*	152.52	157	670	*651*	65.59
160	546	*530*	9.81	165	418	418	0.98	166	426	*414*	52.70	169	386	386	0.74
182	464	464	0.37	184	659	659	0.64	189	608	608	0.97	190	504	504	0.47
211	670	*668*	4.53	213	449	*440*	1.77	214	457	*455*	0.70	215	627	627	1.90
217	635	635	1.00	218	605	605	0.60	219	458	*445*	2.05	220	685	*677*	0.95
225	615	615	0.14	241	827	*782*	26.30	242	803	*765*	96.46	243	739	*713*	9.88
245	520	*507*	3.56	247	481	*443*	104.31	249	707	*684*	31.53	251	484	484	0.24
252	600	600	0.35	254	402	402	5.17	258	543	543	0.88	272	597	597	0.51
274	474	*473*	0.48	275	613	613	1.07	276	526	*523*	0.46	277	575	*569*	0.41
280	665	665	0.71	303	485	*473*	2.33	304	451	451	2.84	305	644	*634*	3.56
306	596	*578*	1.46	309	761	*754*	2.18	332	699	*682*	75.84	333	588	*583*	1.58
336	658	658	0.63	337	531	*506*	6.67	338	632	632	1.12	339	839	839	31.77
340	770	*753*	184.18	343	522	522	0.29	346	432	432	0.18	348	457	457	0.46
355	431	431	0.53	358	588	588	0.21	361	544	544	0.63	363	430	430	2.50
369	504	504	0.46	370	662	662	0.34	391	655	*653*	3.93	392	624	624	1.22
393	655	*654*	0.99	394	507	*487*	2.95	396	691	*687*	1.47	397	636	636	1.72
398	422	*400*	3.66	399	546	*543*	1.90	400	723	723	0.87	419	551	551	0.17
421	589	*550*	77.41	422	729	729	0.63	423	791	*776*	67.25	424	789	*757*	21.17
425	813	*783*	17.20	426	707	707	1.00	427	592	592	0.81	428	584	584	0.99
429	663	*629*	2.33	430	770	*768*	7.14	431	394	394	0.71	436	477	477	0.42
437	578	578	0.38	440	619	619	0.23	452	616	*610*	0.85	455	546	546	0.28
456	676	676	0.57	457	553	553	0.21	458	538	538	0.53	462	373	373	0.12
470	483	483	0.17	481	546	546	0.47	482	656	656	1.61	483	578	578	0.31
484	795	795	0.81	485	622	*615*	1.56	486	692	692	1.44	487	663	663	0.96
489	630	*615*	4.93	490	778	778	1.02	498	508	508	0.14	501	669	669	0.15
504	438	438	0.16	507	527	527	0.21	511	719	719	1.28	512	580	*562*	4.30
513	576	*566*	2.47	514	800	800	1.48	516	801	801	0.79	517	603	*602*	0.92
518	709	709	2.87	519	618	*600*	3.25	520	695	695	2.32	522	431	431	0.27
523	409	409	0.75	524	492	*490*	0.65	528	450	450	0.27	529	421	421	0.29
530	486	486	0.29	540	510	510	0.74								

Table 5.22: All closed instances from class J20.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
34	95	95	0.58	35	103	103	1.11	38	106	106	0.43	48	50	50	0.01
58	63	63	0.01	65	92	92	5.45	70	117	117	1.00	71	58	*56*	0.05
72	50	*49*	0.08	73	59	*58*	0.07	75	24	*23*	0.05	77	46	46	0.02
78	38	38	0.04	80	28	*27*	0.06	81	43	43	0.01	88	36	36	0.02
90	40	40	0.01	128	100	100	0.44	130	98	98	0.34	149	64	64	0.00
150	47	*46*	0.01	154	119	119	15.90	167	52	*50*	0.04	170	63	63	0.01
220	113	113	0.48	246	119	119	1.25								

Table 5.23: All closed instances from class J30.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
4	104	*101*	1.12	12	48	*46*	0.04	13	63	63	0.04	17	57	57	0.02
20	32	*31*	0.02	24	39	39	0.02	32	114	*113*	0.23	33	135	*114*	5.70
37	119	*118*	8.15	38	93	*90*	5.42	40	120	*113*	2.64	41	47	*46*	0.07
42	64	64	0.02	45	56	*54*	0.04	46	51	*47*	0.06	47	46	46	0.02
53	46	46	0.02	57	70	70	0.02	59	58	*55*	0.09	60	47	*46*	0.09
67	130	130	15.89	68	174	174	27.08	71	56	*54*	0.08	75	65	*61*	0.19
76	72	*68*	0.19	77	48	*46*	0.68	78	64	*61*	0.07	79	71	71	0.09
80	65	65	0.03	89	80	80	0.04	102	60	60	0.04	114	42	42	0.01
119	79	79	0.02	123	151	*150*	265.21	124	133	133	1.72	129	145	145	0.29
131	83	83	0.02	133	101	101	0.02	134	59	*57*	0.14	138	96	96	0.03
139	89	*88*	0.03	144	102	102	0.01	149	105	105	0.01	154	134	134	34.96
163	54	*53*	0.15	165	70	*69*	0.18	167	112	112	0.03	168	45	*43*	4.90
170	96	*95*	0.08	173	85	85	0.02	174	60	60	0.03	175	71	*70*	0.03
176	93	93	0.05	195	58	*55*	0.03	204	52	*51*	0.03	224	116	116	0.03
230	116	116	0.02	244	153	153	1.24	247	175	175	0.34				

Table 5.24: All closed instances from class UBO10.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
9	37	37	0.00	16	28	28	0.00	18	45	45	0.00	23	32	32	0.00
26	34	34	0.00	29	33	33	0.00	34	50	50	0.01	38	57	57	0.00
41	39	39	0.00	43	40	40	0.00	47	27	27	0.00	58	31	31	0.00
60	30	30	0.00	75	32	32	0.00	81	59	59	0.00				

Table 5.25: All closed instances from class UBO20.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
4	98	98	0.06	6	108	108	0.08	8	93	93	0.04	10	106	106	0.08
13	92	92	0.01	15	46	*45*	0.02	17	69	69	0.02	20	66	*65*	0.02
21	44	44	0.01	25	39	39	0.01	26	61	61	0.01	28	58	58	0.01
32	86	86	0.05	34	125	125	0.02	40	106	106	0.09	41	62	62	0.01
44	78	78	0.01	46	73	73	0.01	48	88	88	0.02	49	63	63	0.01
54	57	57	0.01	56	56	56	0.01	57	107	107	0.01	60	40	40	0.00
65	119	119	0.03	74	99	99	0.01	82	53	53	0.00	87	75	75	0.00

Table 5.26: All closed instances from class UBO50.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
6	232	*213*	0.48	9	230	*194*	13.81	11	146	*141*	0.16	12	115	115	0.13
13	134	134	0.11	14	168	*153*	0.10	15	105	*99*	0.19	17	112	*109*	0.19
18	164	*163*	0.06	19	166	*156*	0.16	23	161	161	0.05	30	289	289	0.04
31	308	*302*	0.26	34	232	*223*	10.53	36	218	*204*	16.38	37	232	*229*	0.27
40	203	*201*	149.02	41	139	139	0.15	42	148	*147*	0.06	43	100	*98*	0.14
45	187	*181*	0.08	47	196	196	0.08	49	153	*145*	0.11	51	124	124	0.05
52	139	*137*	0.05	54	91	*89*	0.06	55	191	191	0.06	57	133	*132*	0.06
58	182	182	0.04	60	128	128	0.08	61	288	288	0.22	63	240	240	0.72
64	326	326	0.24	65	215	*198*	10.59	67	246	*243*	0.36	68	278	*275*	0.41
71	156	156	0.10	76	162	162	0.09	77	260	260	0.12	78	219	219	0.10
80	298	298	0.10	82	149	149	0.04	84	169	169	0.07	85	190	190	0.04
87	269	269	0.11	88	245	245	0.05	89	218	218	0.05	90	243	243	0.06

Table 5.27: All closed instances from class UBO100.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
11	263	*243*	0.53	12	224	*216*	1.09	13	180	*158*	0.90	14	206	*190*	1.90
16	144	*131*	16.91	18	306	306	0.26	19	200	*177*	4.67	20	209	*201*	0.36
21	262	262	0.21	22	492	492	0.30	27	225	*204*	0.88	36	457	*405*	127.34
38	483	*477*	3.00	39	462	*457*	2.60	41	363	363	0.45	42	359	*358*	0.44
44	491	*483*	0.57	45	407	407	0.38	46	283	*278*	1.19	47	302	*301*	0.32
48	433	433	0.46	50	269	*260*	0.81	51	272	*267*	0.36	53	177	177	0.22
55	247	247	0.25	56	288	288	0.27	57	356	356	0.34	59	256	256	0.21
60	188	188	0.25	61	680	680	3.59	62	540	*526*	17.73	64	538	*533*	2.30
65	451	*433*	284.60	67	459	*402*	39.81	68	540	*538*	1.18	71	514	514	0.49
73	414	*398*	0.57	74	255	*228*	1.46	76	411	411	0.62	77	351	*345*	0.68
78	412	*410*	1.15	79	483	483	0.61	81	453	*452*	0.44	82	571	*568*	0.29
85	497	497	0.24	86	531	531	0.31	87	368	*363*	0.28	88	402	402	0.35
89	374	374	0.23	90	476	476	0.24								

Table 5.28: All closed instances from class UBO200.

Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*	Inst	Best *UB*	Optimal	Best *rt*
11	424	*362*	25.07	14	467	*442*	4.68	15	363	*361*	1.33	16	604	604	1.75
17	470	470	6.51	18	382	*377*	1.17	28	371	371	1.75	30	350	350	1.12
41	571	*533*	13.52	42	721	*712*	3.82	43	653	*642*	1.16	45	522	*514*	1.60
46	572	572	2.01	47	380	*345*	5.18	48	853	853	2.19	49	696	*683*	2.21
50	650	650	1.10	51	581	581	1.53	52	612	612	2.34	53	624	624	1.63
58	689	689	1.50	63	1424	*1422*	27.69	68	1205	*1155*	42.62	69	994	*943*	72.23
71	728	*725*	3.01	72	720	*717*	2.96	73	861	*856*	2.76	74	1176	*1175*	9.77
75	830	*827*	4.86	76	810	*808*	2.32	77	804	*762*	11.94	78	778	*773*	2.31
79	760	*757*	2.25	82	774	774	1.51	83	820	*817*	2.11	84	463	463	1.57
85	592	592	1.27												

Instances with Better Upper Bound

Table 5.29 on page 146 lists all 51 instances for which our methods could find a new upper bound on the makespan, but could not prove the optimality. For each instance the following parameters are given: the class of the instance (Class), the instance number (Inst), the previously best known upper bound (Best *UB*) on the makespan (as reported in Schwindt (2011)), the new best upper bound (New *UB*), and the lowest runtime (Best *rt*) of our methods to find a schedule with the new best upper bound.

5.5 Final Remarks

In this chapter, we evaluated a new approach solving RCPSP and RCPSP/max problems by using cumulative constraints with explanation in a LCG system. For both problems, we used a two-stage process: in the first phase a solution is generated or infeasibility proven (in the case of RCPSP/max), and in the second phase a branch-and-bound algorithm is used for minimising the project duration where the problem is set up with an upper bound on the project duration found from the first solution.

First, we showed that modelling cumulative constraints by decomposition and using LCG is highly competitive on RCPSP problems. We then improved this by using a global cumulative propagator with explanation. Benchmarks from Kolisch and Sprecher (1997) show the strong power of nogoods and VSIDS style search to fathom a large part of the search space. Without building complex specific global propagators or highly specialised search algorithms, we are able to compete with highly specialised RCPSP solving approaches and close 71 open problems.

Second, we used our best approach on RCPSP problems—*i.e.*, the approach us-

Table 5.29: All new *UB* for instances from all classes.

Class	Inst	Best *UB*	New *UB*	Best *rt*	Inst	Best *UB*	New *UB*	Best *rt*	Inst	Best *UB*	New *UB*	Best *rt*
C	61	407	393	417.24	63	380	366	530.73	66	385	368	550.93
	67	367	350	151.22	69	388	380	80.91	70	391	379	305.15
	125	351	316	159.53	152	384	369	569.71	160	394	374	504.98
	218	325	309	148.55	241	430	416	406.44	245	468	453	483.84
	310	277	267	549.74	331	388	381	212.15	336	393	371	429.53
	337	296	287	103.62	339	429	412	234.70				
D	35	420	408	286.63	63	544	524	528.09	158	696	687	390.24
	246	489	463	406.39								
J30	65	163	162	0.15	73	57	53	187.12	151	158	157	303.55
UBO50	3	204	196	130.22	4	253	216	424.01	10	204	192	117.31
UBO100	4	429	410	132.72	7	447	419	364.42	8	435	400	547.80
	10	522	453	343.38	32	485	448	95.06	33	435	418	313.43
	34	488	425	245.22	37	453	426	399.41	40	504	473	97.51
	70	422	410	296.43								
UBO200	2	1000	938	1029.50	3	951	906	242.97	4	1009	963	1477.46
	5	866	852	278.91	6	939	841	311.27	8	998	911	400.51
	33	892	855	351.70	34	931	816	272.33	36	1025	921	391.01
	37	843	798	495.62	39	906	898	505.51	62	853	847	389.05
	67	977	904	566.38	70	1009	972	3102.61				

ing the global cumulative constraint with explanation—to tackle the even harder RCPSP/max problems on three benchmark suites from Schwindt (2011). Experiments on these suites show that our solver is able to find better solutions faster than competing approaches, and prove optimality for many more instances than competing approaches.

In contrast to some previous approaches we used individual propagators for precedence constraints instead of propagators taking all precedence constraints into account at once. This yielded not only weaker propagation, but also slower detection of infeasibility, in particular for instances with a large number of precedences. Hence, our generic search used in the first phase is sometimes slower in finding a first solution than some other problem-specific approaches in the literature. However, the first-phase generic search could be replaced by one of these methods.

Our method closes 573 open RCPSP/max problems and improves a further 51 upper bounds on the project duration of the 58 remaining open problems, according to the best known results given in Schwindt (2011). We note though that the methods from Ballestín *et al.* (2009) and Oddi and Rasconi (2009) may have found better upper bounds on some of these problems, but we could not find a record of

them. Note that our method is highly robust: it proves the best known optimum for each already closed instance in every test. Furthermore, for every open instance in every test set we either close the instance or improve the upper bounds, except for 7 instances, in 4 of which we still regenerate the best known upper bound.

Overall, our generic approach not only is competitive on the considered benchmarks with highly specialised approaches, but also closes in total 644 open problems, improves 78 (51) lower (upper) bounds on the project duration, and is highly robust. The key is to lazily build specialised explanations for propagation which are then used for conflict learning, and to combine the approach with a search that exploits retrieved information from conflict learning.

6

Carpet Cutting — An Application

The carpet cutting problem is a two-dimensional cutting and packing problem in which carpets are cut from a rectangular carpet roll with a fixed roll width and a sufficiently long roll length. The goal is to find a non-overlapping placement of all carpets on the carpet roll, so that the wastage of the carpet roll is minimised while meeting all domain-specific constraints. Moreover, the customers require a cutting solution to be produced within 3 minutes, in order to be usable during the quotation process for estimating the amount of carpet required.

In this chapter, we develop complete solution approaches that decompose the problem into independent subproblems and minimise the wastage for each subproblem separately. Each decomposition uses two global cumulative constraints, one for the roll width and one for the roll length, in order to prune more search space. The cumulative propagation is explained with explanations developed in Chap. 4. On 150 real-world instances provided by the customer we show that our solutions reduce the wastage by more than 35% on average compared to the existing approach.

6.1 Introduction

The carpet cutting problem consists of a set of carpet shapes (also called items or objects) that must be cut from a carpet roll with a fixed roll width and a sufficiently long roll length. The sizes of these objects and the carpet roll are discretised to a granularity of 1cm. For a cutting solution some items can be rotated or even split into several pieces before being placed on the roll if additional constraints are satisfied. The farthest edge of any object from the beginning of the carpet roll determines the last cut across the roll width. Hence, the wastage of a cutting

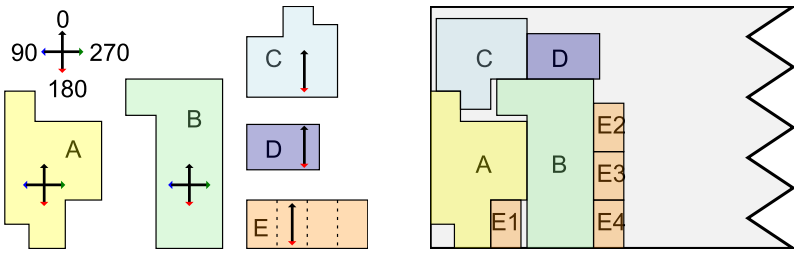


Figure 6.1: Example of a carpet cutting instance.

solution is the wastage from the start of the carpet roll to the last cut. Therefore, minimising the wastage corresponds to minimising the needed roll length.

In this work the carpet shapes are rectilinear polygons of up to 12 sides that can be made up of non-overlapping rectangles, that must be placed orthogonally on the carpet roll, *i.e.*, their edges must be parallel to the borders of the roll. Before the placement of a carpet shape a rotation may be allowed by 90°, 180°, or 270°, *i.e.*, it can be put onto the roll in one of four cardinal directions 0°, 90°, 180°, or 270°. But depending on the pile direction of the carpet there may be restrictions on the which cardinal directions can be used: perhaps only 0°, 180° or perhaps fixed to 0°.

Normally, a carpet shape is cut as a single piece from the carpet roll, but carpet shapes for covering stairs or filling up the remainder of a room are allowed to be cut in several pieces provided that the partition of these carpet shapes satisfy additional constraints which are described later. The joint of carpets for stairs that is then introduced between two adjacent pieces can be hidden between the tread and the riser of the stairs once they are laid. The resulting seams of carpets filling up a room are hidden at the edge of a room. Moreover, these carpet shapes are simple rectangles.

Another complexity of the problem is that carpets have a *pile direction* that may constrain the orientations of some carpet shapes to be dependent on one or another. Clients may also prefer to have the pile direction fixed to ensure an even colour of the carpet when laid relative to a window. Where two carpets join, e.g. at a doorway, the pile direction becomes visible if the two pieces are not laid with a similar pile direction. Therefore, all carpet shapes that are joined together must be arranged pile aligned in the plan. Carpet shapes for stairs must be pile aligned with the pile direction being up the stairs, for safety reasons this ensures that it is harder to slip down the stairs.

Example 6.1. Figure 6.1 shows an example of a carpet cutting instance. On the left side the five carpet shapes A, B, C, D, and E are shown and on the right side their placement on the carpet roll (gray area). The roll is laid out from the left to

the right, *i.e.*, its width is the vertical edge and its length the horizontal one.

On the left-hand side each object contains arrows displaying in which direction the object can be placed where an arrow pointing to the top, left, bottom, or right stands for the direction 0°, 90°, 180°, and 270° respectively. As shown the object A, and B can placed in any direction, but not the objects C, D, and E which must be pile aligned. Moreover, the object E is a carpet for covering four stair steps. The vertical dotted line shows the edge between the tread and riser of two steps. The object E can be split at those edges.

In the placement shown on the right-hand side the objects A and B are placed in the 0° direction whereas the other objects are rotated by 180°. The object E is partitioned in four parts in order to minimise the needed roll length. □

The carpet retailer uses a solution as a base of an on-site cost estimation and ordering process, to submit an offer to customers. The offer should be made in a timely manner and a three-minutes runtime limit is given to the cut-plan optimisation process.

The carpet cutting problem can be characterised as an extension of a two-dimensional orthogonal strip packing (Osp) problem (referred to as a two-dimensional orthogonal open dimension problem by Wäscher *et al.* (2007)) with additional constraints in which a packing of rectangles with minimal wastage is sought. The extensions are the placement constraints between rectangles belonging to the same carpet shape and the partition constraints for carpet shapes covering stairs and filling up the remainder of the room. The side effect of the first constraints is that for those carpet shapes a rotation by 180° and 270° may not be symmetric to a rotation by 0° and 90° respectively.

For Osp and related cutting and packing problems different methods have been applied; a survey can be found in Lodi *et al.* (2002). The different methods can be roughly categorised in these groups: (1) positional placement/reasoning and (2) relational placement/reasoning. The first category includes methods such as the bottom-left rule (Hadjiconstantinou and Christofides 1995, Martello and Vigo 1998) and the discretisation of the large rectangle (Beldiceanu and Carlsson 2001). The second category includes methods that determine the relations (above, under, left, and right) of each pair of rectangles (Pisinger and Sigurd 2007) and the graph-theoretical models (Fekete *et al.* 2007). Our approach includes features of both categories.

A two-dimensional cutting and packing problem can be relaxed into two scheduling problems. One of the problems is projected on the length-axis of the large rectangle and the other on the width-axis of the large rectangle. These relaxations can be

used in order to infer more about possible positions of the items to be laid on the large rectangle and detect infeasibilities of partial solutions earlier.

Constraint programming methods include the global cumulative constraints (Aggoun and Beldiceanu 1993) that models a cumulative scheduling problem, the sweep pruning technique for k-dimensional objects (Beldiceanu and Carlsson 2001) and the geost constraint (Beldiceanu *et al.* 2007b) (modelling k-dimensional objects that can take different shapes). Moreover, special pruning algorithms exist for the cumulative constraint in the case of non-overlapping rectangles (Beldiceanu *et al.* 2008). The sweep algorithm and the geost constraint are specifically designed to model non-overlapping objects with at least two-dimensions. These algorithms demonstrate very good results if the *slack* (the unused space) is small. If the slack is not small then the additional computational effort may not rewarded by the reduction of the search space.

The existing solution used in the field (Pearson *et al.* 1998) uses a combination of heuristic search and dynamic programming in a series of optimisation steps. The algorithm incrementally selects carpet shapes that are placed across the roll considering all alternatives and reduces the overall length of material in a branch-and-bound backtracking search. The algorithm is complex and can be subject to reduced performance when certain rare combinations of heuristic choice lead to inefficiencies of placement. It is incomplete and often uses the full 3 minutes of runtime, but considerably less for smaller problems. It was designed to run on 100MHz tablet PCs with considerably less computing power available than today's processors.

We define two new complete approaches to the carpet cutting problems. The first approach decomposes the problem into multiple instances where all the carpets have fixed dimension and orientation. These subproblems are sequentially solved by an LCG solver (see Chap. 2) maintaining the best solution found overall. Since all dimensions are fixed the constraint propagation is strong. But a disadvantage is there may be many instances for a single problem. The second approach models the orientation of the carpet as a variable and hence reduces the number of instances required for each problem. It can handle problems that the first approach cannot.

6.2 The Carpet Cutting Problem

In the carpet cutting problem there are three different types of carpet shapes:

- *room carpets* that cover rooms, which are made up of a number of rectangular pieces which are constrained to align;

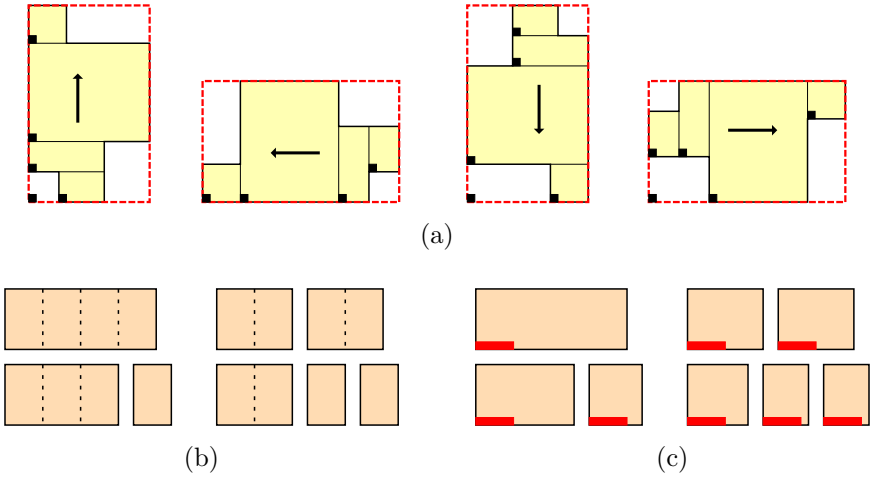


Figure 6.2: The origin of a room carpet and its rectangles in each orientation (a). Possible partitions for a stair carpet (b) and an edge filler carpet (c).

- *stair carpets* that cover stairs, which can be cut into regular pieces and are always rectangular;
- *edge filler carpets* that cover the remainder of a room that is only slightly wider than the width of the carpet. The remainder of the room is covered with multiple narrow pieces cut at any point providing each piece is of a minimum length.

A room carpet is characterised by its set of possible orientations and offsets from its origin to the origin of its rectangles for each orientation. The origin of a room carpet is the bottom left corner of the smallest rectangle that encloses all its rectangles in each orientation. Each rectangle has a width and a length which are given for the 0° orientation. The carpet *origin* is the bottom left corner in each orientation. Where a room is larger in both directions than the width of the carpet, a choice of where the full roll width is aligned is made by the user in advance of the placement optimisation.

Example 6.2. Figure 6.2a shows the room carpet laid out in each orientation. Its smallest enclosing rectangle is displayed with a red-dotted line. The small black squares in each rectangle indicate the origin for the carpet and its rectangles. These pictures show how the offsets from the origin differ for each orientation. □

Stair and edge filler carpets are characterised by their width and length. Each of them may be allowed to be cut in several pieces. Stair carpets are cut with regular breaks between the tread and the riser of two or more steps hence each single piece must cover an integral number of steps.

Edge filler pieces may be cut arbitrarily with irregular length breaks. These shapes can be divided at any position so long as their length is not smaller than a minimal given length. The resulting seams are hidden at the edge of a room. Significant savings in material wastage occur for certain single room carpet orders using this approach. For both kinds of breaks a maximal number of pieces and minimal length of sub-pieces can be given.

Example 6.3. Figure 6.2b shows a stair carpet with 4 pieces and possible partitions, with a maximum of three pieces allowed. Figure 6.2c shows possible partitions for an edge filler carpet with length 200 units, with a minimal length of 50 units (indicated by the bar in the bottom left corner) and a maximum of two cuts. □

A formal specification of an instance I of the carpet cutting problem is defined as follows. We are given 3 sets of disjoint objects:

- $Room$ is a set of room carpets. Each $c \in Room$ is defined by a set of rectangles $c.rect$. For each rectangle $r \in c.rect$ we have a length $r.len$ and width $r.wid$ (in the $0°$ orientation) together with an offset $(r.ox, r.oy)$ from the origin of the room carpet (in the $0°$ orientation). Moreover, each $c \in Room$ is also given a set of allowable orientations $c.ori \subseteq \{0°, 90°, 180°, 270°\}$.

- Str is a set of stair carpets. For each $c \in Str$ we have a width $c.wid$, step length $c.step$ and number of steps $c.n$ as well as a maximum number of pieces $c.pcs$ and minimum steps per piece length $c.min$.

- Edg is a set of edge filler carpets. For each $c \in Edg$ we have a width $c.wid$, length $c.len$ as well as a maximum number of pieces $c.pcs$ and minimum length per piece length $c.min$.

The remaining part of the model is a set $Pile \subseteq Room$ which determines which carpets must be pile aligned, *i.e.*, $c.ori = \{0°, 180°\}$ for each $c \in Pile$, and a roll width RW. Hence, $I = (Room, Str, Edg, Pile, RW)$. Note that all stair and edge filler carpets must be pile aligned, but this constraint can be neglected, since the pile orientations are symmetrical for rectangles as it is for parts of these carpets.

The aim is to find an allowable partitioning $c.part$ of each carpet $c \in Str \cup Edg$ into rectangles, and position (x, y) and allowed orientation for each rectangle r appearing in a room carpet such that: none of the rectangles overlap; each of the rectangles in a room carpet are correctly offset from the origin of the carpet; all pile aligned carpets are aligned in the same orientation, and the roll length RL is minimised.

Figure 6.3 shows the best solution found by our method for a large instance. It reduces the wastage by about 33% in comparison to the current method.

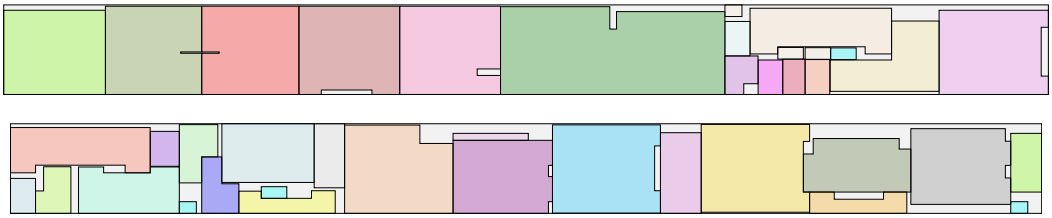


Figure 6.3: A solution (split into two parts) for Cc instance with 34 room carpets (involving 74 rectangles) and 2 stair carpets (involving 7 rectangles). The roll length is about 93m to a granularity of 1cm.

6.3 Static Model

The first model we present, the *static* model, splits the original problem into instances where the orientations and dimensions of each of the rectangular pieces are fixed in advance (*statically known*). This is achieved by fixing rotations of room carpets and fixing the partitions for stair carpets. The advantage of the static model is that it reduces the number of variables required to specify the problem, and gives stronger initial propagation. It reduces the requirements of the global constraints needed to model non-overlap, since dimensions are fixed. It also improves the strength of preprocessing. The obvious disadvantage of the static model is that the number of instances required to specify one original problem may become prohibitive.

To apply the static model we wish to fix the orientation and dimensions of all the rectangles in the problem. To do so we have to split the problem into multiple instances. For many problems in the industrial data the number of instances required is not too large since they are often reasonably constrained.

6.3.1 Dealing with Orientations

Every carpet $c \in Room \setminus Pile$ has an allowable set of orientations in {0°, 90°, 180°, 270°}. We can split an instance I to remove possibilities of different orientations for a carpet c by creating the set of instances $I_o, o \in c.ori$ that are each identical to I except that $c.ori = \{o\}$, and for room carpets we swap the length $r.len$ and width $r.wid$ of the component rectangles if $o \in \{90°, 270°\}$, and update the offsets $(r.ox, r.oy)$ to reflect them from the new origin in this orientation.

If pile aligned carpets are involved in an instance then the instance is split into two instances. In one instance all pile aligned carpets c are fixed to the orientation 0° and in the other to 180°.

Note that before doing this we preprocess instances in order to reduce the possible orientations of carpets:

- For room carpets consisting of one rectangle the orientations 0° and 180° (90° and 270°) are symmetric. If both orientations are given then one of them is removed. For square carpets the orientation is fixed to 0°.
- Some room carpets are too wide for the carpet roll if they are placed in a certain orientations. All those orientations are removed.
- Finally, if all room carpets in one instance that are made of more than one rectangle must be pile aligned then the pile alignment constraint is removed from all of them and their orientation is fixed to 0°, since each solution for the direction 0° is a solution for the direction 180° by rotating the carpet roll and all the placed objects by 180°.

6.3.2 Stair Carpets

Carpets for stairs play an important role for the difficulty of a problem because they can be partitioned in many combinations and introduce symmetries if two parts in the partition have the same length. We can ameliorate the difficulty of stair carpets by avoiding considering all possible partitions by determining "dominated" partitions.

Example 6.4. Suppose a stair carpet covers 15 steps and can be cut into an unlimited number of pieces where each part must consist of at least two steps. Possible partitions are $\{10, 5\}$, $\{10, 3, 2\}$, $\{5, 4, 3, 3\}$, etc. where each multiset represents a partition and the elements express the size in steps of each piece. The total number of possible partitions (incl. the partition $\{15\}$) is 41. □

The partition problem is well studied in number theory. The (generating) function that counts the number of different partitions for a sum n is called the *partition function* (George 1998). This function grows exponentially as the value n increases. For stair carpets an important simplification of the problem arises when we realise that not all partitions need to be considered because some parts of a partition can be broken into smaller pieces which can be laid out in a way identical to the original coarser pieces.

Example 6.5. Consider a stair carpet with possible partitions $\{10, 5\}$ and $\{10, 3, 2\}$. Given a layout for the first partition, the piece of length 5 steps in the first partition can be split into two parts in which one part covers three steps and the other one two steps, thus giving a layout for the second partition. Hence we need not consider laying out the first partition, the partition $\{10, 5\}$ is *dominated* by the partition $\{10, 3, 2\}$. □

Definition 6.1. Let P_1 and P_2 be two different partitions of n (*i.e.*, $\sum P_1 = \sum P_2 = n$). We say $P_2 = \{p_{21} \ldots, p_{2k}\}$ is *dominated* by $P_1 = \{p_{11}, \ldots, p_{1m}\}$ if and only if there is a mapping $\sigma : 1..m \to 1..k$ such that $\forall i \in 1..k : p_{2i} = \sum_{j \in 1..m \text{ where } \sigma(j)=i} p_{1j}$. That is we can further partition P_2 to obtain P_1. Given a set of partitions $\mathbf{P}$ we say $P \in \mathbf{P}$ is *dominating* if it is not dominated by any $P' \in \mathbf{P} - \{P\}$.

It follows that only dominating partitions must be considered during the solution process. We now construct a recursive definition of the number $nd(n, p, k)$ of dominating partitions for a stair carpet of length n steps with maximum number of pieces p and minimal step length k as follows:

$$nd(n, p, k) = nd(n, p, k, k) \ ,$$

where

$$nd(n, p, l, k) = \begin{cases} 0 & \text{if } 0 < n \text{ and } n < l, \\ 0 & \text{if } 0 \wedge p > 0 \text{ and } l \geq 2k, \\ 1 & \text{if } n = 0 \text{ and } p = 0, \\ 1 & \text{if } n = 0,\ p > 0, \text{ and } l < 2k, \\ \sum\limits_{l \leq i \leq n} nd(n - i, p - 1, i, k) & \text{otherwise.} \end{cases}$$

The function $nd(n, p, l, k)$ returns the number of dominating partitions for a carpet of length n, maximal pieces p, minimum length l and minimum original length k. The definition captures the following reasoning. The first case is where there is carpet left but it is smaller that the minimal required length. The second case is when there is no carpet left but there are pieces remaining and one of the earlier pieces (which is at least size l) could be split in two. The third case is where there is no carpet and no pieces left. The fourth case is when there is no carpet left, and more pieces are possible but the longest piece is not large enough to split. The recursive case adds up the possibilities of selecting a piece of size i in the range l to n from a carpet of size n, and determine how many ways to partition the remaining carpet. The remaining subproblem is for a carpet of length $n-i$, with one less piece possible, and a minimum length of i (so we pick pieces in increasing order). The function can be easily modified to return the dominating partitions.

In the customer data the parameter k is either 1 or 2 and the number of steps n in a stair carpet ranges from 1 to 18 and 2 to 15 for $k = 1$ and $k = 2$ respectively. For most of the customer data the number of cuts constraint is not constraining ($\geq n$

Table 6.1: All dominating partitions for various lengths n where the minimal step length k is 2, and maximal pieces is n (so effectively no limit on pieces).

n	partitions	n	partitions	n	partitions
2	$\{2\}$	3	$\{3\}$	4	$\{2,2\}$
5	$\{3,2\}$	6	$\{3,3\}$, $\{2,2,2\}$	7	$\{3,2,2\}$
8	$\{3,3,2\}$, $\{2,2,2,2\}$	9	$\{3,3,3\}$, $\{3,2,2,2\}$	10	$\{3,3,2,2\}$, $\{2,2,2,2,2\}$
11	$\{3,3,3,2\}$, $\{3,2,2,2,2\}$	12	$\{3,3,3,3\}$, $\{3,3,2,2,2\}$, $\{2,2,2,2,2,2\}$	13	$\{3,3,3,2,2\}$, $\{3,2,2,2,2,2\}$
14	$\{3,3,3,3,2\}$, $\{3,3,2,2,2,2\}$, $\{2,2,2,2,2,2,2\}$	15	$\{3,3,3,3,3\}$, $\{3,3,3,2,2,2\}$, $\{3,2,2,2,2,2,2\}$		

when $k = 1$ and $\geq \lfloor n/2 \rfloor$ when $k = 2$), and the total number of dominating partitions is small. This means we can separate the problem into different instances with different fixed (dominating) partitions. Table 6.1 shows the dominating partitions for stair carpets up to 15 steps for $k = 2$. If $k = 1$ then the partition with n parts "1", *i.e.*, $\{1, \ldots, 1\}$ is the only dominating partition for stair carpets covering n steps.

We can split a carpet cutting instance I involving a stair carpet c as follows. For a stair carpet c we determine the set of dominating partitions $\mathbf{P}$ of c and create a new instance $I_P, P \in \mathbf{P}$ where $P = \{p_1, \ldots, p_m\}$ which is identical to I except that the partition function for carpet c is fixed so that $c.part = \{r_1, \ldots, r_m\}$ and the rectangular pieces r_i are constrained as follows: $r_i.wid = c.wid$, $r_i.len = p_i \times c.step$.

Too many dominating partitions

For some cases in the customer data, for example $n = 18$, $k = 1$ and $p = 7$, there are 49 dominating partitions. Splitting into different instances becomes prohibitive when we have to consider other reasons for splitting such as multiple stair carpets, and different room carpet orientations.

When the number of dominating partitions is too large, we modify the partitioning as follows. We consider the partitioning problem with no limit on the number of pieces (or equivalently limit n). For the customer data, the maximal number of dominating partitions that arise with this weakening is 3 (as illustrated by Tab. 6.1). We split into instances using these dominating partitions. This model of course can create a carpet cutting with too many carpet pieces for a regular carpet c. For each rectangle $r \in c.part$ we add a Boolean variable $r.last$ to the model.

We constrain $r.last$ to hold if the rectangle does not have another rectangle $r' \in c.part$ directly to the right (6.1) and ensure that there are at most $c.pcs$ last parts (6.2). These constraints exist for all carpets $c \in Str$:

$$\forall r \in c.part: \quad r.last \leftrightarrow (\forall r' \in c.part \setminus \{r\} : r.x + r.len \neq r'.x \vee r.y \neq r'.y) \quad (6.1)$$

$$\sum_{r \in c.part} r.last \leq c.pcs \ . \quad (6.2)$$

6.3.3 The Model

After handling rotations and stair carpets our original instance I is transformed into a set of *static instances* $\mathbf{I}$ in which all rectangles are fixed in orientation and length and width. If the splitting process created too many instances $\mathbf{I}$ or involved edge filler carpets then we will have to handle the original problem using the dynamic model defined in the next section.

We can now model each static instance $I' \in \mathbf{I}$ reasonably straightforwardly. Let a variable tuple $(r.x, r.y)$ be defined for each rectangle in the instance $Rect = (\bigcup_{c \in Str} c.part) \cup (\bigcup_{c \in Room} c.rect)$ which gives the position of the rectangle on the roll, and variable tuples $(c.x, c.y)$ for each room carpet $c \in Room$. We introduce variable RL to hold the roll length. The constraints of the model are (6.1–6.2) if required, together with:
Each rectangle must be on the roll

$$\forall r \in Rect : 0 \leq r.x \wedge r.x + r.len \leq RL \wedge 0 \leq r.y \wedge r.y + r.wid \leq RW \ . \quad (6.3)$$

Each rectangle in a room carpet must be placed correctly relative to the carpet

$$\forall c \in Room, \forall r \in c.rect : r.x = c.x + r.ox \wedge r.y = c.y + r.oy \ . \quad (6.4)$$

No rectangles overlap which can be modelled with the global constraint `diff2`.

$$\texttt{diff2}([r.x \mid r \in Rect], [r.y \mid r \in Rect], [r.len \mid r \in Rect], [r.wid \mid r \in Rect]) \quad (6.5)$$

For the solver we use, there is no global definition of `diff2`, instead it is decomposed into a disjunction of possibilities where $\prec$ is simply an arbitrary total order imposed on the rectangles.

$$\forall r_1, r_2 \in Rect \text{ s.t. } r_1 \prec r_2 : r_1.x + r_1.len \leq r_2.x \vee r_2.x + r_2.len \leq r_1.x \\ \vee\, r_1.y + r_1.wid \leq r_2.y \vee r_2.y + r_2.wid \leq r_1.y \ . \quad (6.6)$$

This decomposition is very weak, and only propagates if three inequalities are unsatisfiable. In order to get a stronger propagation, two global cumulative constraints are used: one for the roll length and the other one for the roll width. We hence enhance the model with the redundant constraints

$$\texttt{cumulative}([r.x \mid r \in Rect], [r.len \mid r \in Rect], [r.wid \mid r \in Rect], RW) \ , \quad (6.7)$$

$$\texttt{cumulative}([r.y \mid r \in Rect], [r.wid \mid r \in Rect], [r.len \mid r \in Rect], RL) \ . \quad (6.8)$$

The constraints `cumulative` are implemented as global constraints with explanation as described in Chap. 4. They provide much stronger propagation than the decomposed `diff2`. Equation (6.8) also provides strong lower bound reasoning on the objective RL.

In order to find the optimal solution to an original problem instance I we must find the minimal roll length solution for any of the instances into which $\mathbf{I}$ was split.

6.4 Dynamic Model

The static model splits the problem into multiple instances to fix the dimensions of the rectangles. But this can be prohibitive when an original problem splits into very many instances, and it does not give an approach to edge filler carpets. The dynamic model models the problem more directly.

6.4.1 Orientation

For each room carpet c we model its orientation with variable $c.vori$ which takes a value in $c.ori$. We introduce two Boolean variables $c.0or180$ which is true if the carpet is oriented at 0° or 180°, and similarly $c.0or90$.

For each rectangle r we introduce a variable $r.vlen$ to hold its length (after orientation), and similarly a variable to hold its width $r.vwid$, and x offset $r.vox$ and y offset $r.voy$ from the carpet origin. For each carpet c and rectangle $r \in c.rect$ we precalculate two arrays of offsets of r from the carpet origin and each orientation $o \in \{0°, 90°, 180°, 270°\}$ given by $ox_{c,r}[o]$, and $oy_{c,r}[o]$.

The model includes the following constraints for each carpet $c \in Room$:
Enforcing agreement of the orientation and Boolean variables

$$c.0or180 = (c.vori \in \{0°, 180°\}) \ \wedge \ c.0or90 = (c.vori \in \{0°, 90°\}) \ . \quad (6.9)$$

Setting length, width and offsets of each rectangle depending on orientation

$$\begin{aligned} \forall r \in c.rect: \quad & r.vox = ox_{c,r}[c.vori] \ \wedge \ r.voy = oy_{c,r}[c.vori] \\ \wedge \quad & r.vwid = r.len + (r.wid - r.len) \times c.0or180 \qquad (6.10) \\ \wedge \quad & r.vlen = r.wid + (r.len - r.wid) \times c.0or180 \ . \qquad (6.11) \end{aligned}$$

Note that the offset calculation constraints are `element` constraints.

6.4.2 Edge Filler Carpets

Given an edge filler carpet $c \in Edg$ we model this with a set of $c.pcs$ different rectangles $c.part$ (so $|c.part| = c.pcs$). We have to ensure that these pieces are either 0 length (and hence only really pseudo pieces) or reach the minimal length.

$$\forall c \in Edg, \forall r \in c.part : r.vwid = c.wid \wedge (r.vlen = 0 \vee r.vlen \geq c.min) \qquad (6.12)$$

And the sum of the lengths must equal the irregular break length

$$\forall c \in Edg: \quad \sum_{r \in c.part} r.vlen = c.len \ . \qquad (6.13)$$

We can also reason about dominating partitions for irregular breaks. Any partition with a piece r where $r.vlen \geq 2c.min$ and one piece of zero length will be dominated by a partition where r is broken in two. Hence we can add

$$\forall c \in Edg: \ (\exists r \in c.part : r.vlen = 0) \rightarrow (\forall r \in c.part: \ r.vlen < 2c.min) \ . \qquad (6.14)$$

If $c.len \geq 2(c.pcs - 1) \times c.min$ then there can be no zero length pieces since the right hand side of the implication in (6.14) cannot be satisfied at the same time as (6.13), hence in this case we can simplify (6.12).

6.4.3 The Model

The set of rectangles is $Rect = \bigcup_{c \in Room} r.rect \cup \bigcup_{c \in Str \cup Edg} c.part$. We assume that for each stair piece $r.vlen = r.len$ and $r.vwid = r.wid$. The constraints of the model are: (6.1–6.2) if required, (6.3–6.8) with $r.len$ replaced by $r.vlen$ and $r.wid$ replaced with $r.vwid$, (6.9–6.11) and (6.12–6.14) if required.

6.5 Refining the Models

The basic model can be further enhanced in order to improve the propagation, reduce the model size, and strengthen the reasoning and the conflict-driven search in the LCG solver.

6.5.1 Variable Views

Variable views (Schulte and Tack 2005) are a form of variable aliasing. Suppose $y = ax + c$ where a and c are constants, then rather than creating a new variable for y use a view to compute information about the (view) variable y from the real variable x. This refinement (views) is particularly useful for LCG solvers since it improves learning and the generation of strong explanations.

For a fixed orientation room carpet c we can replace the variables $r.x$ and $r.y$ by views on $c.x$ and $c.y$ for all $r \in c.rect$ using (6.4). For non-fixed orientation carpets c we can use views to define $r.vlen$ and $r.vwid$ for $r \in c.rect$ using (6.10) and (6.11).

6.5.2 Disjunction and Better `diff2` Decomposition

In all carpet cutting problems the roll width is narrow in comparison to some wide carpets, so that many other carpets cannot horizontally overlap with these wide carpets. We say these carpets are *in disjunction*. Carpets that are in disjunction with all others can be placed at the beginning of the roll. We denote this as the disj refinement.

It is simple to check whether two rectangles are disjunctive, but not if we consider room carpets made up of several rectangles and possible rotations of them. Algorithm 6.1 gives an overview of detecting carpets that are in disjunction with all other carpets. Each carpet $c \in Room \cup Str \cup Edg$ has the additional attributes $minDm$ and $minCp$ which are defined as follows:

$$c.minDm = \begin{cases} \min_{o \in c.ori} c.maxW[o] & \text{if } c \in Room, \\ c.wid & otherwise \end{cases}$$

$$c.minCp = \begin{cases} \min_{o \in c.ori} c.minW[o] & \text{if } c \in Room, \\ c.wid & otherwise \end{cases}$$

where $c.minW[o]$ and $c.maxW[o]$ are the minimal accumulated width of the room carpet c across its length and the width of the tightest enclosing rectangle of c in the orientation o. Both values are exemplified in Fig. 6.4. Note that for a room

Algorithm 6.1: Detection of disjunctive carpets.

Input: *carp* an array of carpets $c \in Room \cup Str \cup Edg$ sorted in increasing order of minimal diameter $c.minDm$.

Data: RW the roll width and $minRectWid$ minimal width of any rectangle in $Rect$.

Output: *disj* an array of *true* and *false* values where a *true* entry for index means that the carpet $carp[i]$ is disjunctive to all other carpets, and *false* in all other cases.

```
1  for i = 1 to n do disj[i] = true;
2  for i = 1 to n do
3      if disj[i] = false then continue;
4      if i ≠ 1 and carp[i].minDm + carp[1].minDm ≤ RW then
5          disj[i] = false;
6          continue;
7      if carp[i].minCp + minRectWid > RW then continue;
8      for j = 1 to n do
9          if i = j and carp[i] ∈ Str ∪ Edg then
10             continue;
11         if carp[i].minDm + carp[j].minDm ≤ RW then
12             disj[i] = false;
13             disj[j] = false;
14             break;
15         if carp[i].minCp + carp[j].minCp > RW then
16             continue;
           // Expensive check
17         if is_disjunctive(carp[i], carp[j]) then
18             disj[i] = false;
19             disj[j] = false;
20             break;
```

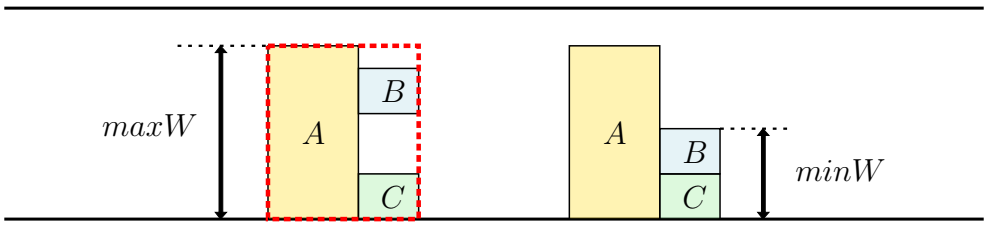


Figure 6.4: The room carpet is depicted in one orientation. It is made up by the rectangles A, B, and C. On the left side the room carpet is shown with its tightest enclosing rectangle (dotted lines) and the width $maxW$ of this rectangle. On the right side the minimal accumulated width $minW$ of the carpet is indicated across the length of the carpet.

carpet c consisting of one rectangle it holds $c.minDm = c.minCp$. For each carpet $carp[i]$—where i is the increment of the outer loop (lines 9–20)—the algorithm checks whether it can overlap with the carpet with the smallest $minDm$ (line 4). If not then the next test checks whether the carpet is in disjunction with the thinnest rectangle (line 6), hence disjunctive with all other carpets. If neither case holds for the carpet $carp[i]$ then the algorithm individually tests if $carp[i]$ is disjunctive with the others $carp[j]$ (lines 8–17). In line 11 the test simply checks if $carp[i]$ and $carp[i]$ can be placed on top of each other whereas in line 14 the test checks if their accumulated minimal width exceeds the roll width. In the first case, $carp[i]$ is not disjunctive and in the second case it is disjunctive with $carp[j]$. If neither case holds then at least one of the carpets must be a room carpet made up of more than one rectangle. Then the algorithm calls the procedure *is_disjunctive* which ultimately checks if both carpets are in disjunction.

Procedure *is_disjunctive* checks the satisfiability of the problem $(\{c_i, c_j\}, \emptyset, \emptyset, \{c_k \mid carp[k] \in Pile\}, RW)$ with an additional constraint on the roll length where c_i and c_j are room carpets created from $carp[i]$ and $carp[j]$ respectively as follows for $k \in \{i, j\}$:

$carp[k] \in Room$**:** $c_k = carp[k]$;

$carp[k] \in Str$**:** The stair carpet is transformed to the room carpet c_k with $c_k.rect = \{r_k\}$, $c_k.ori = \{0°\}$, $r_k.wid = carp[k].wid$, $r_k.len = carp[k].step \cdot carp[k].min$, $r_k.ox = 0$ and $r_k.oy$;

$carp[k] \in Edg$**:** The edge-filler carpet is transformed to the room carpet c_1 with $c_k.rect = \{r_k\}$, $c_k.ori = \{0°\}$, $r_k.wid = carp[k].wid$, $r_k.len = carp[k].min$, $r_k.ox = 0$ and $r_k.oy$.

The roll length RL is constrained to $RL < c_i.maxL[c_i.vori] + c_j.maxL[c_j.vori]$ where $c.maxL[o]$ is a variable modelling the length of the smallest rectangle enclosing the room carpet c if c is placed in orientation o. Clearly, if c_i and c_j are disjunctive then the constraint cannot be satisfied. We solved this problem with a refined dynamic model with the addition of the constraint on RL.

In the worst case Alg. 6.1 performs $\mathcal{O}(n^2)$ disjunctive checks where n is the number of carpets. If the algorithm could not determine, with tests only, whether two carpets are disjunctive then a final time-consuming check is done. The implemented algorithm can take up to 12 seconds for large instances to compute all carpets that are in disjunction with all other carpets.

We can use disjunction to improve the `diff2` decomposition (diff2). If a pair of rectangles r_1 and r_2 cannot overlap along the width of the carpet role we replace

the body of (6.6) by the conjunction of two reified constraints on the same Boolean variable.

$$(b \rightarrow r_1.x + r_1.len \leq r_2.x) \ \wedge \ (\neg b \rightarrow r_2.x + r_2.len \leq r_1.x) \ .$$

Thus, we reduce the number of Boolean variables needed from 4 in (6.6) to 1. For two rectangles we check two cases of disjunctions. The first one simply checks if $lb(r_1.vwid) + lb(r_2.vwid) > RW$. The second one is based on compulsory parts (see Lahrichi 1982) along the roll width and the minimal distance of the bottom and top edge of a rectangle to the top and bottom border of the carpet roll, respectively. The distance from the bottom $r.bottom$ and top $r.top$ edge of a rectangle r are defined as

$$r.bottom = \begin{cases} \max\limits_{o \in c.ori} \{RW + oy_{c,r}[o] - c.maxW[o]\} & \text{if } r \in c.part \text{ and } c \in Room, \\ RW - r.vwid & otherwise \end{cases}$$

$$r.top = \begin{cases} \min\limits_{o \in c.ori} \{oy_{c,r}[o] + r.len[o]\} & \text{if } r \text{ belongs to a room carpet } c, \\ r.vwid & otherwise \end{cases}$$

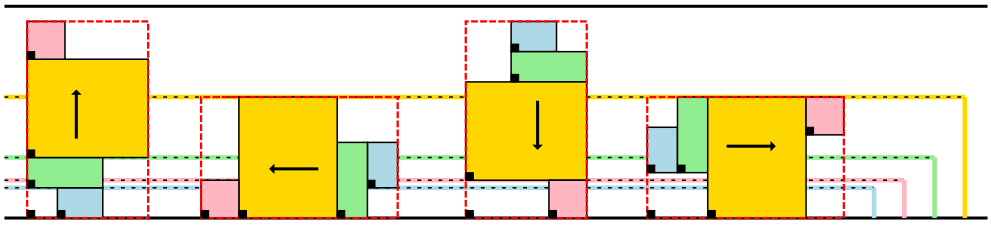
where $r.len[o]$ is the length of the rectangle placed in orientation o. If $r.bottom < r.top$ then r creates a compulsory part covering the y-coordinates in $[\, r.bottom \,..\, r.top \,]$.

Example 6.6. Consider the room carpet from Ex. 6.2 shown in Fig. 6.2a on page 153. Figure 6.5 shows the carpet laid in each orientation on a carpet roll where in Fig. 6.5a and 6.5b the room carpet is directly placed on the bottom and top border of the carpet roll, respectively. Each rectangle r of the room carpet is differently coloured and the same coloured horizontal line represents $r.top$ or $r.bottom$. Only the golden rectangle creates a compulsory part. □

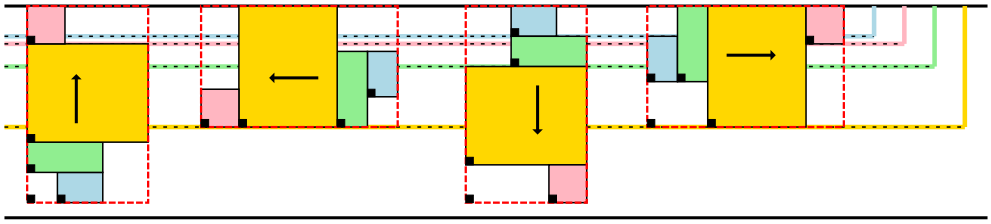
Assume that $r_1.top - r_1.bottom > r_2.top - r_2.bottom$ for two rectangles. If the following condition holds then both rectangles are in disjunction:

$$r_1.bottom < r_1.top \ \wedge \ r_1.bottom < r_2.top \ \wedge \ r_2.bottom < r_1.top \ . \qquad (6.15)$$

The first conjunct checks if the rectangle r_1 creates a compulsory part along the roll width and the two remaining conjuncts hold if the rectangle r_2 intersects with the compulsory part of r_1 when r_2 is directly laid out at the bottom and top border of the carpet roll. Figure 6.6 gives two examples.

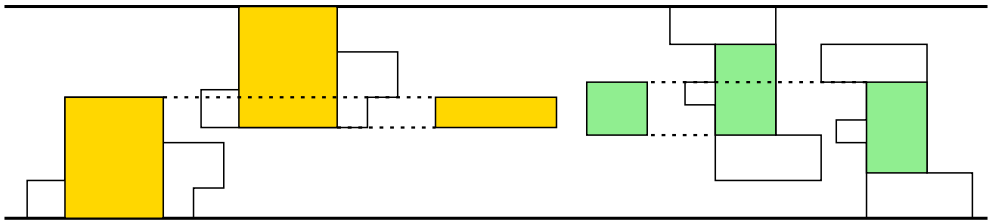


(a) Coloured horizontal lines represent the values of *top*.

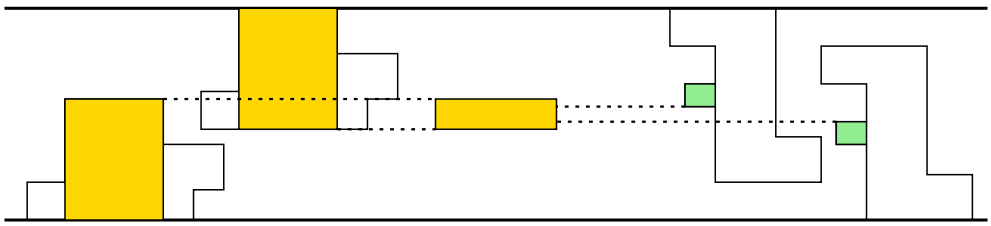


(b) Coloured horizontal lines represent the values of *bottom*.

Figure 6.5: Calculation of the values *top* and *bottom* for the green, blue, gold, and pink rectangle which belong the one room carpet. In each sub-figure the coloured horizontal lines represent the value *top* or *bottom* for the same coloured rectangle.



(a) Both coloured rectangles create a compulsory parts which overlap each other.



(b) The left coloured rectangle creates a compulsory part which intersects with the right rectangle if the right one is laid out close to the roll borders.

Figure 6.6: A carpet roll is shown where two room carpets are directly placed on the top and the bottom border of the roll. In the middle of the roll, for each room carpet a possible compulsory part of the coloured rectangle is drawn. Each sub-figure shows one example of two rectangles which satisfy (6.15).

6.5.3 Symmetry Breaking Constraints

In the model symmetries can occur between rectangles that have the same size, *i.e.*, length and width. A symmetry means that those rectangles can be interchanged in any solution. In order to remove symmetries an order on the origins of the rectangles can be posted.

The most common case for symmetries occurs for pieces of stair carpets. We assume a function $same(r, r')$ which (statically) tests if two rectangles have the same dimensions, are not rotatable and are not part of a room carpet with more than one rectangle. For refinement sym we add a lexicographic ordering on $(r.y, r.x)$ for rectangles that are the same. Symmetry breaking can also considerably simplify the definition of $r.last$ for stair carpets $c \in Str$ since we only need to consider the lexicographically least member of each symmetric group that appears in the partition $c.part$. Finally we can enforce that the pieces of an edge filler carpet are ordered in length. We could consider looking for more complex symmetries like entire room carpets that are identical but they do not occur in the customer data.

6.5.4 Forbidden Gaps

Forbidden gaps (Simonis and O'Sullivan 2008) are areas between a rectangle and a long edge (either from another rectangle or a boundary) that are too small to accommodate any part of other rectangles. In this work, we forbid these gaps between rectangles that do not belong to room carpets with multiple rectangles, and the borders of the carpet roll as follows.

Let *gap* be the minimal width of any rectangle. In the y direction (fbg y) We consider how many rectangles (multiples of *gap*) might fit between the considered object edge and the border of the carpet roll and restrict the values that $r.y$ can take for a rectangle r:

none: if $ub(RW) < lb(r.y) + gap$. We set $r.y$ to 0.

one: if $ub(RW) < lb(r.y) + 2 \cdot gap$. The rectangle is aligned with either the top or the bottom of the roll, *i.e.*, $r.y = 0$ or $r.y = RW - r.vwid$.

two: in all other cases. We enforce that the rectangle is placed either at the bottom border of the roll, at least *gap* units from the borders, or at the top border, *i.e.*, $r.y = 0$, $r.y \geq gap \wedge r.y \leq RW - r.vwid - gap$, or $r.y = RW - r.vwid$.

Similarly, we impose forbidden gaps (fbg x) for the left and right border of the roll.

6.6 Search Strategies

To solve a carpet cutting problem instance I with our approaches we need to solve a series of instances $\mathbf{I}$ determined by splitting. The generic algorithm first attempts to find a good solution for each $I' \in \mathbf{I}$ and then uses the best solution found as an upper bound on roll length, and searches for an optimal solution of each $I' \in \mathbf{I}$ ordered accordingly to the first solution we found for them. During this process the upper bound is always the best solution found so far.

The two phase approach has two benefits. First it means that domain sizes of variables in the optimisation search are much smaller. Because lazy clause generation generates a Boolean representation of the size of the initial domain size this makes the optimisation search much more efficient. Second, the first phase ranks the split instances on the likelihood of finding good solutions, so usually later instances in the optimisation phase are quickly found to be unable to lead to a better solution.

6.6.1 First Solution Generation

The goal of the first search is to quickly generate a first solution that gives a good upper bound on the carpet roll length. We examine each split instance in $\mathbf{I}$ in turn. We order the split instances by the partitions of regular stair carpets, examining partitions with fewer pieces before partitions with more pieces, and otherwise breaking ties arbitrarily.

We use a simple sequential search on each split instance. We treat the room carpets first, in decreasing order of total area. First we assign a horizontal or vertical orientation for all room carpets by fixing the $c.0 or 180$, which fixes the dimensions of each rectangle. Then we fix the orientation by fixing $c.0 or 90$. We then fix the lengths of edge filler carpets. We next determine $c.x$ for all room carpets c, and then determine each $c.y$ again in decreasing area order. Finally we place each stair carpet rectangle by fixing $r.x$ and then $r.y$ treating each rectangle in input order.

6.6.2 Minimisation

A hybrid sequential/activity based search is used to find optimal solutions. We first fix the orientations of each room carpet as we did in the first-solution search. Then we switch to the activity-based VSIDS search, which concentrates on variables which are involved in lots of recent failures.

For the activity-based search, we use a geometric restart policy on the number of node failures in order to make the search more robust. The restart base and factor are 128 failures and 2.0, respectively.

6.7 Experiments

The experiments were carried out on a 64-bit machine with an Intel(R) Pentium(R) D processor with 3.4 GHz clock and Ubuntu 9.04. For each original problem instance I an overall 3 minutes runtime limit was imposed for calculating carpets that are in disjunction with all other carpets if the refinement disj is used, finding a first solution and minimising the roll length for all split instances $\mathbf{I}$.

The G12/FDX solver from the G12 Constraint Programming Platform (Stuckey *et al.* 2005) was used as the Lcg solver. We also experimented with the G12 Fd solver using search more suitable for Fd (placement of the biggest carpets first). The latter could only optimally solve 7 instances compared to 76 for Lcg using the same search. This shows that Lcg is vital for solving the problem to prune substantial parts of the search space.

Dynamic versus Static Model

Table 6.2 compares the static and dynamic model as well as the current solution approach on the instances which the static model can handle (126 of 150). It shows the number of instances solved optimally ("opt."), the sum of the best first solutions found for each instance ("init. ΣRL"), the sum of the best solutions found for each instance ("ΣRL") and the area of wastage ("wast."), *i.e.*, for one instance $RL \times RW - \sum_{c \in Rect} c.len \times c.wid$, relatively to the wastage created by the current method as well as the total runtime to solve all instances ("Σrt."). The static approach solves one more problem and its first solutions are better than for the dynamic approach. In total, a better first solution was generated for 55 instances. Where applicable the static approach is preferable.

The existing method finds, but does not prove, 27 optimal solutions. It was tested by the client IF Computer GmbH on a Dell Latitude D820 with an Intel(R) Core(TM) Duo processor T2400 processing with 1.86 GHz clock. The times marked (†) for the existing approach are the sum of times when the best solution was found. Since it cannot prove optimality for the majority of instances the existing method uses the whole 3 minutes. The new approach results in an improvement of wastage of over 33%.

Refinements

Table 6.3 presents the impact of different refinements on the dynamic models. The entry × means that the refinement was used. We compare the different refinements with the same features as before.

Table 6.2: Comparison between dynamic and static approach.

approach	opt.	init. ΣRL	ΣRL	wast.	Σrt.
dynamic	92/126	171,645	160,536	66.5%	6,247s
static	93/126	168,270	160,399	65.9%	6,946s
Current method	27/126	-	167,668	100%	7,450s†

Table 6.3: Results of different refinements.

disj	views	diff2	sym	fbg x	fbg y	opt.	init. ΣRL	ΣRL	wast.	Σrt.
						86/150	232,181	221,542	67.9%	12,721s
×						88/150	232,075	221,521	67.8%	12,360s
	×					89/150	232,181	221,248	66.9%	11,999s
		×				89/150	232,181	221,240	66.9%	11,980s
			×			99/150	232,181	221,344	67.2%	9,933s
				×		88/150	232,181	221,596	68.1%	12,295s
					×	88/150	232,181	221,399	67.4%	12,302s
				×	×	89/150	232,181	221,060	66.3%	12,385s
×	×	×	×	×	×	**106/150**	232,075	**220,775**	**65.2%**	**9,290s**
Current method						30/150	-	230,795	100%	8,988s†

The change in number of optimally solved instances clearly illustrates the important of symmetry breaking for proving optimality. Variable views and forbidden gaps have a minor impact on proving optimality.

We can see a tradeoff in the refinements. Most make it harder to find solutions, but reduce the search space required to prove optimality. When applying all refinements we solve the most instances, and generate solutions with minimal total length, since the new optimal solutions make up for unsolved problems where we found worse solutions.

6.8 Final Remarks

In this chapter we developed an approach to carpet cutting that can find and prove the optimal solution for real-world instances under a strict time limit. After a preprocessing phase the approach decomposes an instance into several subproblems. These subproblems are modelled as either static or dynamic models which use two cumulative constraints. In the static model the length and the width of carpet shapes are fixed whereas in the dynamic model these parameters could be variable. In the last case, the cumulative propagator that considers flexible processing times

of activities, flexible resource usages of activities, and flexible resource capacities, is used. Each subproblem is optimally solved with an Lcg solver using a branch-and-bound algorithm for minimising the wastage and activity-based search heuristic. Once all subproblems are optimally solved then the global optimum is found.

On 150 real-world instances, we showed that our approach reduces the wastage by more than 34% on average in comparison with the existing heuristic approach. The power of the approach comes from the combination of careful modelling of the stair breaking constraints to eliminate symmetries and dominated solutions, and the use of Lcg to drastically reduce the time to prove optimality.

7 Conclusion

This chapter first summarises the presented work and then gives an outlook for future research issues.

7.1 Summary

In the first two chapters, we introduced the basic solving scheme for combinatorial satisfaction and optimisation problems. This scheme is based on a depth-first tree search with backtracking which is enriched with advanced technologies to skip exploration of inconsistent subtrees. In detail, decisions are interleaved with constraint propagation, which removes inconsistent values from the domains of variables. If the search reaches a conflict node then the conflict analysis deduces a nogood. In the remaining search the nogood is propagated which leads to further pruning of inconsistent parts of the search tree. In our experiments we used a search which takes information gathered during the conflict learning to guide the search.

In the third chapter, we developed new incremental algorithms for satisfaction and implication for unit two variable per inequality constraints (Utvpi). These algorithms are based on new theoretical results about reasoning in Utvpi systems of these constraints, and have the best-known asymptotic time complexity. Experiments show that they are faster on sparse Utvpi systems.

Moreover, we built non-incremental algorithms for implication checking and generation with the best-known asymptotic time complexity and a special explanation generation algorithm for minimal unsatisfiable Utvpi subsets and minimal implicants. Overall, we show that the satisfaction and implication problems for Utvpi systems have the same asymptotic time complexity as those ones for difference constraints.

The following chapter is dedicated to strong explanations for propagation algorithms of cumulative resource constraints. First, explanations are built for the standard algorithms time-table and edge-finding, taking account of flexible start times of activities. Second, the explanations for the time-table algorithms are extended with additional consideration of flexible processing times of activities, flexible resource usages of activities, and flexible resource capacities. Moreover, we show the additional complexity and control that comes with these explanations.

The fifth and sixth chapters show the effect of these explaining cumulative propagators on the generic solving process of two basic scheduling problems and one industrial placement problem.

In the fifth chapters, we considered the resource-constrained project scheduling problem and its extension with generalised precedence constraints for minimising the project duration. Our best complete solution approaches are based on basic models and a generic conflict-driven search, which are solved by a lazy clause generation (Lcg) solver. On well-established benchmark sets from the operations research community, our simple and generic approach outperforms the state-of-the-art complete methods and incomplete methods for problems with generalised precedence constraints. Moreover, many open problems could be optimally solved or the lower or upper bound on the project duration improved.

In the sixth chapters, we considered the industrial problem of carpet cutting, which is a two-dimensional placement problem. We developed a complete approach that takes all domain-specific constraints into account. It first preprocesses an instance and then decomposes it into several subproblems which are optimally solved by an Lcg solver, each in turn. The decomposition exploits domain-specific characteristics, e.g. dominating partitions.

Our complete methods using a basic model already outperform the existing incomplete methods on industrial instances. Further model refinements yielded even better solutions. All models use a cumulative constraint for each dimension. Thus, this placement problem shows that not only scheduling problems benefit from the explaining cumulative propagators.

7.2 Outlook

We developed explanations for the cumulative propagation. In certain situations, the propagator has multiple options for explaining its propagation. In this thesis, the choice of explanation is simply based on the input order of the activity. But this might not be the best choice for all cases. If the choice could be based on the

characteristic of the activities or the activities counter of the related variables then it could lead to a larger reduction of the search space.

We proposed explanations for the edge-finding algorithm, but have not yet evaluated them. The question is, whether they improve the solution process just as for the time-table algorithms, since one explanation potentially includes many activities. Thus, a resulting nogood might not be so reusable anymore.

Constraint propagators in an LCG solver can ask for the initial and current domains of the variables. These bounds are used by propagators to generate the strongest explanations, but the initial domains may include values that the search already has proven as globally invalid. If the propagator uses these values in the explanation then the explanation is not a strongest one with respect to the tightest domain that contains no globally invalid values. If an LCG solver provides information about these tightest domain, which are the domains after the constraint propagation in the root node, to the propagators then strongest explanations can always be generated, potentially yielding more reduction of the search space.

We considered two basic project scheduling problems with the objective to minimise the project duration. This objective function is advantageous for constraint-based approaches, because a new upper bound on the project duration has an immediate impact on the upper bound on the start time variables of all activities, *i.e.*, it significantly reduces the search space. However, scheduling problems often consider other measurements such as linear objective functions, e.g. the total weighted tardiness. Normally, a new bound on those measurements only has a small effect on the reduction of the search space, but not for linear programming based approaches. A hybridisation of these methods could be beneficial.

The scheduling problems herein have a size of at most 200 activities, which is considered a small size for some industries where problems with at least 1000 activities are normal. Our complete methods cannot efficiently handle such large problems. Future research could deal with how to incorporate them with incomplete methods where they repeatedly solve smaller subproblems. Especially, a combination with the large neighbourhood search seems to be fruitful for large scheduling problems (Shaw 2011).

Both of our scheduling approaches use individual propagators for precedence constraints which can result in bad queuing behaviour. A global precedence propagator—taking all precedence constraints into account at once—can resolve this problem in LCG solvers and might speed up the solution process.

We also considered the carpet cutting problem. In order to reduce the complexity of an instance, a preprocessing step was applied. The preprocessing step can be

extended to detect the following situation: the biggest carpets that are in disjunction with each other might be aligned across the carpet roll length so that all small carpets whose projection can vertically overlap with at least one of them can be placed above and below them without overlapping. The resulting placement is an optimal one which can be fixed in the preprocessing step.

Our solution approach may be further improved by using an explaining version of the global non-overlapping constraint `diff2` instead of the primitive non-overlapping constraint for each pair of rectangles.

Experiments for the carpet cutting problem have shown that static symmetry breaking constraints are essential for proving optimality. But for some instances the solution process was significantly slowed down, especially for finding the first solutions. This means that the symmetry breaking constraints pruned the solutions that the search could find quickly. The use of dynamic symmetry breaking constraints may resolve this problem.

In this dissertation, we have substantially improved the state-of-the-art on combinatorial resource-constrained scheduling by developing generic explanations for the propagation of cumulative resource constraint and using the generic framework of lazy clause generation. We believe that this success will trigger further investigations of generic methods using conflict learning in the scheduling area.

Bibliography

Achterberg, T (2009). SCIP: solving constraint integer programs. *Mathematical Programming Computation*, **1**:1–41.

Aggoun, A and Beldiceanu, N (1993). Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, **17**(7):57–73.

Apt, KR (2003). *Principles of Constraint Programming*. Cambridge University Press, Cambridge, UK.

Armando, A, Mantovani, J, and Platania, L (2009). Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer (STTT)*, **11**:69–83.

Artigues, C, Demassey, S, and Néron, E, editors (2008). *Resource-constrained Project Scheduling*. Wiley-ISTE.

Ball, T and Jones, RB, editors (2006). *Proceedings of Computer Aided Verification – CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Ballestín, F, Barrios, A, and Valls, V (2009). An evolutionary algorithm for the resource-constrained project scheduling problem with minimum and maximum time lags. *Journal of Scheduling*, pages 1–15. Doi:10.1007/s10951-009-0125-9.

Baptiste, P and Demassey, S (2004). Tight LP bounds for resource constrained project scheduling. *OR Spectrum*, **26**(2):251–262.

Baptiste, P and Le Pape, C (2000). Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, **5**(1-2):119–139.

Baptiste, P, Le Pape, C, and Nuijten, W (1999). Satisfiability tests and time-bound adjustments for cumulative scheduling problems. *Annals of Operations Research*, **92**:305–333.

Baptiste, P, Le Pape, C, and Nuijten, W (2001). *Constraint-Based Scheduling*. Kluwer Academic Publishers, Norwell, MA, USA.

Barrett, CW, Sebastiani, R, Seshia, SA, and Tinelli, C (2009). *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 12, pages 825–885. IOS Press.

Bartusch, M, Möhring, RH, and Radermacher, FJ (1988). Scheduling project networks with resource constraints and time windows. *Annals of Operations Research*, **16**(1):199–240.

Bedworth, DD and Bailey, JE (1982). *Integrated Production, Control Systems: Management, Analysis, and Design*. John Wiley & Sons, Inc., New York, NY, USA.

Beldiceanu, N and Carlsson, M (2001). Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In T Walsh, editor, *Proceedings of Principles and Practice of Constraint Programming – CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin / Heidelberg.

Beldiceanu, N, Carlsson, M, Demassey, S, and Petit, T (2007a). Global constraint catalogue: Past, present and future. *Constraints*, **12**:21–62.

Beldiceanu, N, Carlsson, M, and Poder, E (2008). New filtering for the cumulative constraint in the context of non-overlapping rectangles. In L Perron and MA Trick, editors, *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2008*, volume 5015 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin / Heidelberg.

Beldiceanu, N, Carlsson, M, Poder, E, Sadek, R, and Truchet, C (2007b). A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In Bessière (2007), pages 180–194.

Bellman, R (1958). On a routing problem. *Quarterly of Applied Mathematics*, **16**(1):87–90.

Berthold, T, Heinz, S, Lübbecke, M, Möhring, R, and Schulz, J (2010). A constraint integer programming approach for resource-constrained project scheduling. In Lodi *et al.* (2010), pages 313–317.

Bessière, C, editor (2007). *Proceedings of Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Błażewicz, J, Lenstra, JK, and Rinnooy Kan, AHG (1983). Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, **5**:11–24.

Bofill, M, Nieuwenhuis, R, Oliveras, A, Rodríguez-Carbonell, E, and Rubio, A (2008a). The barcelogic SMT solver. In A Gupta and S Malik, editors, *Proceedings of Computer Aided Verification – CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298. Springer Berlin / Heidelberg.

Bofill, M, Nieuwenhuis, R, Oliveras, A, Rodríguez-Carbonell, E, and Rubio, A (2008b). A write-based solver for SAT modulo the theory of arrays. In A Cimatti and RB Jones, editors, *Proceedings of Formal Methods in Computer-Aided Design – FMCAD 2008*, pages 1–8. IEEE Press.

Borralleras, C, Lucas, S, Navarro-Marset, R, Rodríguez-Carbonell, E, and Rubio, A (2009). Solving non-linear polynomial arithmetic via sat modulo linear arithmetic. In RA Schmidt, editor, *Proceedings of Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 294–305. Springer Berlin / Heidelberg.

Bozzano, M, Bruttomesso, R, Cimatti, A, Junttila, T, Ranise, S, van Rossum, P, and Sebastiani, R (2005). Efficient satisfiability modulo theories via delayed theory combination. In Etessami and Rajamani (2005), pages 443–454.

Brucker, P, Drexl, A, Möhring, R, Neumann, K, and Pesch, E (1999). Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, **112**(1):3–41.

Brucker, P and Knust, S (2000). A linear programming and constraint propagation-based lower bound for the rcpsp. *European Journal of Operational Research*, **127**(2):355–362.

Brummayer, R and Biere, A (2009). Boolector: An efficient SMT solver for bit-vectors and arrays. In S Kowalewski and A Philippou, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2009*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer Berlin / Heidelberg.

Bryant, RA, Kroening, D, Ouaknine, J, Seshia, SA, Strichman, O, and Brady, B (2009). An abstraction-based decision procedure for bit-vector arithmetic. *International Journal on Software Tools for Technology Transfer (STTT)*, **11**:95–104.

Carlier, J and Pinson, E (2004). Jackson's pseudo-preemptive schedule and cumulative scheduling problems. *Discrete Applied Mathematics*, **145**(1):80–94.

Caseau, Y and Laburthe, F (1996). Cumulative scheduling with task intervals. In MJ Maher, editor, *Logic Programming: Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming – JICSLP 1996*, pages 363–377. The MIT Press.

Cesta, A, Oddi, A, and Smith, SF (2002). A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, **8**(1):109–136.

Cherkassky, BV and Goldberg, AV (1996). Negative-cycle detection algorithms. In J Díaz and MJ Serna, editors, *Proceedings of the European Symposium on Algorithms – ESA 1996*, volume 1136 of *Lecture Notes in Computer Science*, pages 349–363. Springer.

Choi, CW, Harvey, W, Lee, JHM, and Stuckey, PJ (2006). Finite domain bounds consistency revisited. In A Sattar and Bh Kang, editors, *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 49–58. Springer Berlin / Heidelberg.

Cotton, S and Maler, O (2006). Fast and flexible difference constraint propagation for DPLL(T). In A Biere and CP Gomes, editors, *Proceedings of Theory and Applications of Satisfiability Testing – SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 170–183. Springer Berlin / Heidelberg.

Damay, J, Quilliot, A, Sanlaville, E, Damay, J, Quilliot, A, and Sanlaville, E (2007). Linear programming based algorithms for preemptive and non-preemptive rcpsp. *European Journal of Operational Research*, **182**(3):1012–1022.

Davis, M, Logemann, G, and Loveland, D (1962). A machine program for theorem-proving. *Communications of the ACM*, **5**:394–397.

Davis, M and Putnam, H (1960). A computing procedure for quantification theory. *Journal of the ACM*, **7**:201–215.

De Reyck, B and Herroelen, W (1998). A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, **111**(1):152–174.

Dechter, R, Meiri, I, and Pearl, J (1991). Temporal constraint networks. *Artificial Intelligence*, **49**:61–95.

Demeulemeester, E and Herroelen, W (1992). A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management Science*, **38**(12):1803–1818.

Demeulemeester, EL and Herroelen, WS (1997). New benchmark results for the resource-constrained project scheduling problem. *Management Science*, **43**(11):1485–1492.

Demeulemeester, EL and Herroelen, WS (2002). *Project Scheduling: A Research Handbook*. Kluwer Academic Publishers.

Dijkstra, EW (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, **1**:269–271.

Dorndorf, U, Pesch, E, and Phan-Huy, T (2000). A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalised precedence constraints. *Management Science*, **46**(10):1365–1384.

Dutertre, B and de Moura, L (2006). A fast linear-arithmetic solver for DPLL(T). In Ball and Jones (2006), pages 81–94.

El-Kholy, AO (1996). *Resource Feasibility in Planning*. Ph.D. thesis, Imperial College, University of London.

Erschler, J and Lopez, P (1990). Energy-based approach for task scheduling under time and resources constraints. In *2nd International Workshop on Project Management and Scheduling*, pages 115–121.

Etessami, K and Rajamani, S, editors (2005). *Proceedings of Computer Aided Verification – CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Fekete, SP, Schepers, J, and van der Veen, JC (2007). An exact algorithm for higher-dimensional orthogonal packing. *Operations Research*, **55**(3):569–587.

Fest, A, Möhring, RH, Stork, F, and Uetz, M (1999). Resource-constrained project scheduling with time windows: A branching scheme based on dynamic release dates. Technical Report 596, Technische Universität Berlin.

Feydy, T (2010). *Constraint Programming: Improving Propagation*. Ph.D. thesis, The University of Melbourne.

Feydy, T, Schutt, A, and Stuckey, PJ (2008). Global difference constraint propagation for

finite domain solvers. In S Antoy and E Albert, editors, *Proceedings of on Principles and Practice of Declarative Programming – PPDP 2008*, pages 226–235. ACM.

Feydy, T and Stuckey, PJ (2009). Lazy clause generation reengineered. In Gent (2009), pages 352–366.

Ford, LR and Fulkerson, DR (1962). *Flows in Networks*. Princeton University Press.

Franck, B, Neumann, K, and Schwindt, C (2001). Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling. *OR Spectrum*, **23**(3):297–324.

Fränzle, M, Herde, C, Teige, T, Ratschan, S, and Schubert, T (2007). Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, **1**(3-4):209–236.

Frigioni, D, Marchetti-Spaccamela, A, and Nanni, U (1998). Fully dynamic shortest paths and negative cycle detection on digraphs with arbitrary edge weights. In G Bilardi, GF Italiano, A Pietracaprina, and G Pucci, editors, *Proceedings of European Symposium on Algorithms – ESA 1998*, volume 1461 of *Lecture Notes in Computer Science*, pages 320–331. Springer.

Ganesh, V and Dill, DL (2007). A decision procedure for bit-vectors and arrays. In W Damm and H Hermanns, editors, *Proceedings of Computer Aided Verification – CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer Berlin / Heidelberg.

Gaschnig, J (1979). *Performance measurement and analysis of certain search algorithms*. Ph.D. thesis, Carnegie-Mellon University.

Gent, IP, editor (2009). *Proceedings of Principles and Practice of Constraint Programming – CP 2009*, volume 5732 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

George, AE (1998). *The Theory of Partitions*. Cambridge University Press.

Grimes, D and Hebrard, E (2010). Job shop scheduling with setup times and maximal time-lags: A simple constraint programming approach. In Lodi *et al.* (2010), pages 147–161.

Grimes, D, Hebrard, E, and Malapert, A (2009). Closing the open shop: Contradicting conventional wisdom. In Gent (2009), pages 400–408.

Hadjiconstantinou, E and Christofides, N (1995). An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, **83**(1):39–56.

Haralick, RM and Elliott, GL (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, **14**(3):263–313.

Hartmann, S and Kolisch, R (2000). Experimental evaluation of state-of-the-art heuris-

tics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, **127**(2):394–407.

Harvey, W and Stuckey, PJ (1997). A Unit Two Variable Per Inequality Integer Constraint Solver for Constraint Logic Programming. In *The 20th Australasian Computer Science Conference (Australian Computer Science Communications)*, pages 102–111. Sydney, Australia.

Heinz, S and Schulz, J (2011). Explanations for the cumulative constraint: An experimental study. In PM Pardalos and S Rebennack, editors, *Proceedings of Experimental Algorithms – SEA 2011*, volume 6630 of *Lecture Notes in Computer Science*, pages 400–409. Springer Berlin / Heidelberg.

Hentenryck, PV, Saraswat, V, and Deville, Y (1998). Design, implementation, and evaluation of the constraint language cc(fd). *The Journal of Logic Programming*, **37**(1-3):139–164.

Hooker, JN (2005). A hybrid method for the planning and scheduling. *Constraints*, **10**:385–401.

Hooker, JN (2007). *Integrated Methods for Optimization*. Springer.

Hooker, JN and Yan, H (2002). A relaxation of the cumulative constraint. In P Van Hentenryck, editor, *Proceedings of Principles and Practice of Constraint Programming – CP 2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 686–691. Springer Berlin / Heidelberg.

Horbach, A (2010). A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, **181**:89–107.

Jaffar, J, Maher, MJ, Stuckey, PJ, and Yap, RHC (1994). Beyond finite domains. In A Borning, editor, *Proceedings of Principles and Practice of Constraint Programming – PPCP 1994*, volume 874 of *Lecture Notes in Computer Science*, pages 86–94. Springer Berlin / Heidelberg.

Johnson, DB (1977). Efficient algorithms for shortest paths in sparse networks. *J. ACM*, **24**(1):1–13.

Junker, U (2004). QuickXPlain: preferred explanations and relaxations for over-constrained problems. In McGuinness and Ferguson (2004), pages 167–172.

Jussien, N (2003). The versatility of using explanations within constraint programming. Research Report 03-04-INFO, École des Mines de Nantes, Nantes, France. URL `http://www.emn.fr/jussien/publications/jussien-RR0304.pdf`.

Jussien, N and Barichard, V (2000). The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133. Singapore.

Jussien, N, Debruyne, R, and Boizumault, P (2000). Maintaining arc-consistency within

dynamic backtracking. In R Dechter, editor, *Proceedings of Principles and Practice of Constraint Programming – CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 249–261. Springer Berlin / Heidelberg.

Jussien, N and Lhomme, O (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, **139**(1):21–45.

Katsirelos, G and Bacchus, F (2005). Generalized nogoods in CSPs. In MM Veloso and S Kambhampati, editors, *Proceedings on Artificial Intelligence – AAAI 2005*, pages 390–396. AAAI Press / The MIT Press.

Kelley, JE, Jr. (1963). *The criticial-path method: Resources planning and scheduling*, chapter 21, pages 347–365. Prentice-Hall.

Kolisch, R (1996). Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, **90**(2):320–333.

Kolisch, R (2001). *Make-to-order assembly management*. Springer.

Kolisch, R and Hartmann, S (2006). Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, **174**(1):23–37.

Kolisch, R, Schwindt, C, and Sprecher, A (1998). *Project Scheduling: Recent Models, Algorithms and Applications*, chapter Benchmark instances for project scheduling problems, pages 197–212. Kluwer Academic Publishers.

Kolisch, R and Sprecher, A (1997). PSPLIB – A project scheduling problem library. *European Journal of Operational Research*, **96**(1):205–216.

Kolisch, R, Sprecher, A, and Drexl, A (1995). Characterization and generation of a general class of resource-constrained project scheduling problems. *Management Science*, **41**(10):1693–1703.

Kowalski, R (1979). Algorithm = logic + control. *Communications of the ACM*, **22**:424–436.

Kroening, D and Strichman, O (2008). *Decision Procedures: An Algorithmic Point of View*. Springer.

Laborie, P (2005). Complete MCS-based search: Application to resource constrained project scheduling. In LP Kaelbling and A Saffiotti, editors, *Proceedings of Artificial Intelligence – IJCAI 2005*, pages 181–186. Professional Book Center.

Lahiri, SK and Musuvathi, M (2005). An efficient decision procedure for UTVPI constraints. In B Gramlich, editor, *Proceedings of Frontiers of Combining Systems – FroCoS 2005*, volume 3717 of *Lecture Notes in Computer Science*, pages 168–183. Springer Berlin / Heidelberg.

Lahiri, SK, Nieuwenhuis, R, and Oliveras, A (2006). SMT techniques for fast predicate abstraction. In Ball and Jones (2006), pages 424–437.

Lahrichi, A (1982). Scheduling: the notions of hump, compulsory parts and their use in cumulative problems. *C. R. Acad. Sci., Paris, Sér. I, Math.*, **294**(2):209–211.

Land, AH and Doig, AG (1960). An automatic method of solving discrete programming problems. *Econometrica*, **28**(3):497–520.

Lecoutre, C, Sais, L, Tabary, S, and Vidal, V (2007). Nogood recording from restarts. In MM Veloso, editor, *Proceedings of Artifical Intelligence – IJCAI 2007*, pages 131–136. Morgan Kaufmann Publishers Inc.

Liess, O and Michelon, P (2008). A constraint programming approach for the resource-constrained project scheduling problem. *Annals of Operations Research*, **157**(1):25–36.

Lodi, A, Martello, S, and Monaci, M (2002). Two-dimensional packing problems: A survey. *European Journal of Operational Research*, **141**(2):241–252.

Lodi, A, Milano, M, and Toth, P, editors (2010). *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2010*, volume 6140 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Mackworth, AK (1977). Consistency in networks of relations. *Artificial Intelligence*, **8**(1):99–118.

Manna, Z and Zarba, CG (2003). Combining decision procedures. In B Aichernig and T Maibaum, editors, *Proceedings of Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer Berlin / Heidelberg.

Marriott, K, Nethercote, N, Rafeh, R, Stuckey, PJ, Garcia de la Banda, M, and Wallace, MG (2008). The design of the Zinc modelling language. *Constraints*, **13**(3):229–267.

Marriott, K and Stuckey, PJ (1998). *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, MA, USA.

Martello, S and Vigo, D (1998). Exact solution of the two-dimensional finite bin packing problem. *Management Science*, **44**(3):388–399.

McGuinness, DL and Ferguson, G, editors (2004). *Proceedings of Artifical intelligence – AAAI 2004*. AAAI Press / The MIT Press. ISBN 0-262-51183-5.

Meir, O and Strichman, O (2005). Yet another decision procedure for equality logic. In Etessami and Rajamani (2005), pages 31–37.

Mercier, L and Van Hentenryck, P (2008). Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, **20**(1):143–153.

Miné, A (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, **19**(1):31–100.

Mohr, R and Henderson, TC (1986). Arc and path consistency revisited. *Artificial Intelligence*, **28**(2):225–233.

Moskewicz, MW, Madigan, CF, Zhao, Y, Zhang, L, and Malik, S (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of Design Automation Conference – DAC 2001*, pages 530–535. ACM, New York, NY, USA.

Neumann, K and Schwindt, C (1997). Activity-on-node networks with minimal and maximal time lags and their application to make-to-order production. *OR Spectrum*, **19**:205–217.

Nieuwenhuis, R and Oliveras, A (2005). DPLL(T) with exhaustive theory propagation and its application to difference logic. In Etessami and Rajamani (2005), pages 305–309.

Nieuwenhuis, R, Oliveras, A, and Tinelli, C (2005). Abstract DPLL and abstract DPLL modulo theories. In F Baader and A Voronkov, editors, *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning – LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer Berlin / Heidelberg.

Nieuwenhuis, R, Oliveras, A, and Tinelli, C (2006). Solving SAT and SAT Modulo Theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, **53**(6):937–977.

Nuijten, WPM (1994). *Time and Resource Constrained Scheduling*. Ph.D. thesis, Eindhoven University of Technology.

Oddi, A and Rasconi, R (2009). Iterative flattening search on RCPSP/max problems: Recent developments. In A Oddi, F Fages, and F Rossi, editors, *Proceedings of Recent Advances in Constraints – CSCLP 2008*, volume 5655 of *Lecture Notes in Computer Science*, pages 99–115. Springer Berlin / Heidelberg.

Ohrimenko, O and Stuckey, PJ (2008). Modelling for lazy clause generation. In J Harland and P Manyem, editors, *Proceedings of Fourteenth Computing: The Australasian Theory Symposium (CATS 2008)*, pages 27–37. Australian Computer Society, Darlinghurst, Australia, Australia.

Ohrimenko, O, Stuckey, PJ, and Codish, M (2007). Propagation = lazy clause generation. In Bessière (2007), pages 544–558.

Ohrimenko, O, Stuckey, PJ, and Codish, M (2009). Propagation via lazy clause generation. *Constraints*, **14**(3):357–391.

Pearson, C, Birtwistle, M, and Verden, AR (1998). Reducing material wastage in the carpet industry. In *Proceedings of Applications of Prolog – INAP 98*, pages 88–99.

Pisinger, D and Sigurd, M (2007). Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS Journal on Computing*, **19**(1):36–51.

Pnueli, A, Rodeh, Y, Shtrichman, O, and Siegel, M (1999). Deciding equality formulas by small domains instantiations. In N Halbwachs and D Peled, editors, *Proceedings of Computer Aided Verification – CAV 1999*, volume 1633 of *Lecture Notes in Computer Science*, pages 687–688. Springer Berlin / Heidelberg.

Refalo, P (2004). Impact-based search strategies for constraint programming. In MG Wallace, editor, *Proceedings of Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer Berlin / Heidelberg.

Régin, JC (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of Artificial Intelligence – AAAI 1994*, pages 362–367. AAAI Press, Menlo Park, CA, USA.

Schulte, C and Stuckey, PJ (2005). When do bounds and domain propagation lead to the same search space? *ACM Transactions on Programming Languages and Systems*, **27**:388–425.

Schulte, C and Stuckey, PJ (2008). Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems*, **31**(1):1–43.

Schulte, C and Tack, G (2005). Views iterators for generic constraint implementations. In P van Beek, editor, *Proceedings of Principles and Practice of Constraint Programming – CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 817–821. Springer Berlin / Heidelberg.

Schutt, A, Feydy, T, Stuckey, PJ, and Wallace, MG (2009). Why cumulative decomposition is not as bad as it sounds. In Gent (2009), pages 746–761.

Schutt, A, Feydy, T, Stuckey, PJ, and Wallace, MG (2010). Solving the resource constrained project scheduling problem with generalized precedences by lazy clause generation. URL `http://arxiv.org/abs/1009.0347`.

Schutt, A, Feydy, T, Stuckey, PJ, and Wallace, MG (2011a). Explaining the cumulative propagator. *Constraints*, **16**(3):250–282.

Schutt, A and Stuckey, PJ (2010). Incremental satisfiability and implication for UTVPI constraints. *INFORMS Journal on Computing*, **22**(4):514–527.

Schutt, A, Stuckey, PJ, and Verden, AR (2011b). Optimal carpet cutting. In J Lee, editor, *Proceedings of Principles and Practice of Constraint Programming – CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin / Heidelberg.

Schutt, A and Wolf, A (2010). A new $\mathcal{O}(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In D Cohen, editor, *Proceedings of Principles and Practice of Constraint Programming – CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 445–459. Springer Berlin / Heidelberg.

Schutt, A, Wolf, A, and Schrader, G (2006). Not-first and not-last detection for cumulative scheduling in $\mathcal{O}(n^3 \log n)$. In Umeda *et al.* (2006), pages 66–80.

Schwindt, C (1995). ProGen/max: A new problem generator for different resource-constrained project scheduling problems with minimal and maximal time lags. WIOR 449, Universität Karlsruhe, Germany.

Schwindt, C (1998a). A branch-and-bound algorithm for the resource-constrained project duration problem subject to temporal constraints. WIOR 544, Universität Karlsruhe, Germany.

Schwindt, C (1998b). *Verfahren zur Lösung des ressourcenbeschränkten Projektdauerminimierungsproblems mit planungsabhängigen Zeitfenstern*. Shaker-Verlag.

Schwindt, C (2011). Project generator ProGen/max and PSP/max-library. URL `http://www.wior.uni-karlsruhe.de/LS_Neumann/Forschung/ProGenMax/rcpspmax.html`. Last access at the date of 28 June 2011.

Sebastiani, R (2007). Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, **3**(3-4):141–224.

Sellmann, M and Kadioglu, S (2008). Dichotomic search protocols for constrained optimization. In Stuckey (2008), pages 251–265.

Seshia, S, Subramani, K, and Bryant, R (2007). On solving Boolean combinations of UTVPI constraints. *Journal of Satisfiability, Boolean Modelling and Computation*, **3**:67–90.

Shaw, P (2011). Constraint programming and local search hybrids. In P Van Hentenryck and M Milano, editors, *Hybrid Optimization*, volume 45 of *Springer Optimization and Its Applications*, pages 271–303. Springer New York. ISBN 978-1-4419-1644-0.

Simonis, H and O'Sullivan, B (2008). Search strategies for rectangle packing. In Stuckey (2008), pages 52–66.

Sitzmann, I and Stuckey, PJ (2000). O-trees: a constraint based index structure. In M Orlowska, editor, *Proceedings of Australasian Database Conference – ADC2000*, pages 127–135. IEEE Press.

Smith, TB and Pyle, JM (2004). An effective algorithm for project scheduling with arbitrary temporal constraints. In McGuinness and Ferguson (2004), pages 544–549.

Somogyi, Z, Henderson, F, and Conway, T (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, **29**(1–3):17–64.

Stuckey, PJ, editor (2008). *Proceedings of Principles and Practice of Constraint Programming – CP 2008*, volume 5202 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Stuckey, PJ, García de la Banda, MJ, Maher, MJ, Marriott, K, Slaney, JK, Somogyi, Z, Wallace, MG, and Walsh, T (2005). The G12 project: Mapping solver independent models to efficient solutions. In M Gabbrielli and G Gupta, editors, *Proceedings of Logic Programming – ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 9–13. Springer Berlin / Heidelberg.

Tillmann, N and de Halleux, J (2008). Pex-white box test generation for .NET. In B Beckert and R Hähnle, editors, *Proceedings of Tests and Proofs – TAP 2008*,

volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin / Heidelberg.

Tseitin, GS (1968). On the complexity of derivation in propositional calculus. *Studies in mathematics and mathematical logic*, **Part II**:115–125. Translated from Russian.

Umeda, M, Wolf, A, Bartenstein, O, Geske, U, Seipel, D, and Takata, O, editors (2006). *Proceedings of Declarative Programming for Knowledge Management – INAP 2005*, volume 4369 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Vilím, P (2005). Computing explanations for the unary resource constraint. In R Barták and M Milano, editors, *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2005*, volume 3524 of *Lecture Notes in Computer Science*, pages 396–409. Springer Berlin / Heidelberg.

Vilím, P (2009). Edge finding filtering algorithm for discrete cumulative resources in $\mathcal{O}(kn \log n)$. In Gent (2009), pages 802–816.

Vilím, P (2011). Timetable edge finding filtering algorithm for discrete cumulative resources. In T Achterberg and J Beck, editors, *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 2011*, volume 6697 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg.

Wäscher, G, Haußner, H, and Schumann, H (2007). An improved typology of cutting and packing problems. *European Journal of Operational Research*, **183**:1109–1130.

Wolf, A and Schrader, G (2006). $\mathcal{O}(n \log n)$ overload checking for the cumulative constraint and its application. In Umeda *et al.* (2006), pages 88–101.

Zhang, L, Madigan, CF, Moskewicz, MH, and Malik, S (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of Computer Aided Design – ICCAD 2001*, pages 279–285. IEEE Press, Piscataway, NJ, USA.