Exact Integer Programming Approaches to Sequential Instruction Scheduling and Offset Assignment

Inaugural-Dissertation

zur

Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultät

der Universität zu Köln

vorgelegt von

Sven Mallach

aus Duisburg

Köln 2015

Berichterstatter:	Prof. Dr. Michael Jünger
(Gutachter)	Prof. Dr. Peter Marwedel
Tag der mündlichen Prüfung:	15.01.2015

Zusammenfassung

Die vorliegende Dissertation berichtet über die wesentlichen Konzepte und Resultate wissenschaftlicher Studien zur exakten Lösung zweier $\mathcal{NP}$-schwerer Compiler-Optimierungsprobleme, *Instruction Scheduling* und *Offset Assignment*, mittels ganzzahliger linearer Programmierung. Sie ist das Ergebnis mehrjähriger Forschung als wissenschaftlicher Mitarbeiter am Lehrstuhl von Michael Jünger in Köln, mit einer besonderen Ausrichtung darauf, mathematische Optimierungsverfahren auf praktische Problemstellungen aus dem Bereich der technischen Informatik anzuwenden.

Die beiden behandelten Probleme sind im Wesentlichen völlig unabhängig voneinander, beschäftigen sich aber beide mit der Zuteilung von beschränkt zur Verfügung stehenden Ressourcen und treten während der Erzeugung von Maschinencode innerhalb eines Compilers auf. Instruction Scheduling behandelt die Zuteilung von Taktzyklen zu Instruktionen mit dem Ziel, die Gesamtausführungszeit aller Instruktionen zu minimieren. Diese Zuteilung muss Datenabhängigkeiten, Latenzbedingungen und Ressourcenbeschränkungen berücksichtigen. Beim Offset Assignment geht es um Speicherlayouts von Programmvariablen und den effizienten Einsatz von Adressregistern für Zugriffe auf diese Variablen, so dass der erforderliche Zusatzaufwand in Form von expliziten Adressberechnungen minimiert wird. Im Gegensatz zum Instruction Scheduling, das Bestandteil nahezu jedes Compilers ist, tritt das Offset Assignment Problem hauptsächlich bei Compilern für spezialisierte Prozessorarchitekturen auf.

Instruction Scheduling ist ein bereits sehr intensiv studiertes Problem, zu dem diverse exakte und heuristische Verfahren entworfen und experimentell analysiert wurden. Diese Arbeit konzentriert sich auf das *Basic-Block Instruction Scheduling Problem* für *Single-Issue Prozessoren.* Basic Blocks sind Programmfragmente mit einem einzigen Einstiegs- und Ausstiegspunkt. Es müssen daher alle Instruktionen eines Basic Blocks ausgeführt werden, bevor der Kontrollfluss zu einem anderen Basic Block übergeht. Single-Issue Prozessoren sind in der Lage, pro Taktzyklus mit der Ausführung exakt einer neuen Instruktion zu beginnen. Eine Reihe von Techniken zur Vorbehandlung von Basic Block Instanzen wurden in der Literatur vorgestellt. Sie werden, mit einem Schwerpunkt auf aktuellere Verfahren seit dem Jahr 2000, in dieser Arbeit ausgiebig diskutiert, denn sie führten zu einem Constraint Programming Ansatz im Jahr 2006, der etwa 350000 Instanzen optimal lösen konnte, wobei einige dieser Instanzen bis zu 2500 Instruktionen beinhalten. Der letzte Versuch, das Problem mittels ganzzahliger Programmierung zu lösen, datiert hingegen aus einer Zeit vor Veröffentlichung der jüngsten Vorbehandlungsmethoden. Es erwies sich bei sehr restriktiven Latenzbedingungen als erfolgreich, konnte aber hunderte

der soeben benannten Instanzen, die große und stärker variierende Latenzen beinhalten, nicht lösen. Des Weiteren basieren nahezu alle bisher vorgestellten Verfahren auf sogenannten zeitindizierten Formulierungen, bei denen Entscheidungsvariablen eine explizite Zuweisung von Instruktionen zu Taktzyklen modellieren. Die vorliegende Arbeit beschreibt hingegen einen vollkommen neuen Ansatz basierend auf dem *Linearen Ordnungsproblem*, das ein bereits sehr gut studiertes kombinatorisches Optimierungsproblem ist. Die neuen Modelle führen zu einer alternativen Charakterisierung der zulässigen Lösungen des Basic-Block Instruction Scheduling Problems. Sie ermöglichen den Einsatz von Branch-and-Cut Algorithmen, die auch größere Instanzen lösen können. Die Formulierungen werden außerdem durch zusätzliche Ungleichungen erweitert, die als Schnittebenen verwendet werden können. Kombiniert mit den Methoden zur Vorbehandlung, die zum Teil ebenfalls noch erweitert und verbessert werden, kann die entwickelte Implementierung vergleichbare Ergebnisse wie der Constraint Programming Ansatz erzielen. Dieses Ziel zu erreichen hat einige Jahre in Anspruch genommen. Die vorliegende Arbeit berichtet daher nicht nur über die entwickelten Modelle, sondern auch über diverse Ideen und Nebenprodukte, die in dieser Zeit entstanden und die inspirierend sein bzw. anderen Wissenschaftlern helfen könnten, selbst wenn sie andere Lösungsverfahren einsetzen möchten.

Die Ausgangssituation bzgl. des Offset Assignment Problems war eine andere, da insbesondere exakte Verfahren vor den in dieser Arbeit vorgestellten eher rar waren. Das Offset Assignment Problem kam in den neunziger Jahren auf und wird in diversen Varianten betrachtet, die von theoretischer und praktischer Bedeutung sind. In der einfachsten Variante wird angenommen, dass ein Prozessor lediglich ein einzelnes Adressregister und sehr limitierte Möglichkeiten zur Adressierung von Programmvariablen ohne Verursachung zusätzlichen Aufwands bietet. Selbst für diese einfachste Variante, die auch als Baustein für komplexere dient, ist das Berechnen einer Optimallösung jedoch bereits $\mathcal{NP}$-schwer und somit wurde das Problem im Wesentlichen im Sinne einer heuristischen Lösung studiert. Die wenigen exakten Lösungsverfahren waren nicht in der Lage, mittelgroße Instanzen zu lösen, so dass die tatsächliche Güte heuristischer Lösungen für eine lange Zeit kaum bekannt war. Auch hier zeigte sich die Untersuchung der kombinatorischen Struktur der verschiedenen Problemvarianten als Schlüsselansatz, um Branch-and-Cut Verfahren entwerfen zu können, die von bekanntem Wissen über verwandte kombinatorische Optimierungsprobleme profitieren. Die Implementierung zur Lösung der einfachsten Problemvariante war dann die erste, die die große Mehrheit von etwa 3000 Instanzen eines Standard-Benchmarks optimal lösen konnte. Im Anschluss konnten zunächst, in Zusammenarbeit mit Roberto Castañeda Lozano, zusätzliche Techniken eingebunden werden, die es dem Ansatz erlaubten, auch über den Einsatz mehrerer Adressregister zu optimieren. Erfreulicherweise konnten die Verfahren dann sogar noch weiter verallgemeinert werden, um mit flexibleren Möglichkeiten zur Adressierung von Programmvariablen umgehen zu können. Auf diese Weise beantwortet die vorliegende Arbeit nicht nur die Frage, wie groß der zusätzliche Adressierungsaufwand ist, wenn Heuristiken verwendet werden, sondern liefert auch erste Resultate, die es erlauben, die Auswirkungen flexiblerer Adressierungsmöglichkeiten auf die Laufzeit und Größe realer Anwendungen zu analysieren.

Abstract

The dissertation at hand presents the main concepts and results derived when studying the optimal solution of two $\mathcal{NP}$-hard compiler optimization problems, namely *instruction scheduling* and *offset assignment*, by means of integer programming. It is the outcome of several years of research as an assistant at Michael Jünger's computer science chair in Cologne, with the particular aim to apply exact mathematical optimization techniques to real-world problems arising in the domain of technical computer science.

The two problems studied are rather unrelated apart from the fact that they both take place during the machine code generation phase of a compiler and deal with the handling of limited resources. Instruction scheduling is about the assignment of issue clock cycles to instructions in the presence of precedence, latency, and resource constraints such that the total time needed to execute all the instructions is minimized. Offset assignment deals with storage layouts of program variables and the efficient use of address registers for accesses to these variables. The objective is to employ specialized instructions in order to minimize the overhead caused by address computations. While instruction scheduling needs to be carried out by almost every present compiler irrespective of the processor architecture, the offset assignment problem occurs mainly in compilers for highly specialized processor designs.

Instruction scheduling is a well-studied field where several exact and heuristic approaches have been developed and experimentally evaluated in the past. In this thesis, we concentrate on the *basic-block instruction scheduling problem* for *single-issue processors*. Basic blocks are program fragments with no side-entrances and -exits, i.e., every instruction of a basic block needs to be executed before the control flow may leave it and enter another basic block. Single-issue processors are capable of starting the execution of exactly one instruction per clock cycle. A number of techniques to preprocess instances of the basic-block instruction scheduling problem were proposed in the literature and are, with emphasis on the more recent ones that arose since the year 2000, thoroughly reviewed in this thesis. They finally led to a constraint programming approach in 2006 that was shown to solve about $350,000$ instances to optimality and where some of these instances comprised up to about $2,500$ instructions. The last attempt to tackle the problem using integer programming however dates to a time prior to the publication of the latest preprocessing advances. While being successful on a set of instances that impose very restrictive latency constraints, it was shown to be unable to solve hundreds of instances from the aforementioned benchmark set that comprises also large and varying latencies.

In addition, the previous integer programming models were almost all based on so-called time-indexed formulations where decision variables model an explicit assignment of instructions to clock cycles. In this thesis, a completely different and novel approach is taken based on the *linear ordering problem*, a well-studied combinatorial optimization problem. The new models lead to alternative characterizations of the feasible solutions to the basic-block instruction scheduling problem. These facilitate the employment of advanced integer programming methodologies, in particular the design of branch-and-cut algorithms that can handle larger instances. The formulations are further extended by additional inequalities that can be used as cutting planes. Combined with the preprocessing routines that are partially extended and improved as well, the respective solver implementation eventually turned out to be competitive to the constraint programming method. Reaching this point has taken some years and this thesis presents not only the derived models but also several ideas and byproducts that arose in the meantime, and that can help and inspire researchers even if they aim at the application of different solution methodologies.

The starting point regarding the offset assignment problem was a different one because especially exact solution approaches were rather rare prior to the models presented in this thesis. The offset assignment problem arose in the 1990s and is considered in several variants that are of theoretical and practical interest. In the simplest one, a processor is assumed to provide only a single address register and only very restricted possibilities to avoid address computation overhead. However, even this simplest variant, that may serve as a building block for the more complex ones, is already $\mathcal{NP}$-hard and has been studied mainly from a heuristic point of view. The few existing exact solution approaches were not capable to solve moderately sized instances so that the quality of heuristic solutions relative to the optimum was hardly known at all. Again, the inspection of the combinatorial structure of the various problem variants turned out to be the key for designing branch-and-cut implementations that can profit from knowledge about related combinatorial optimization problems. The implementation targeting the simple problem variant was the first capable to optimally solve the majority of about $3,000$ instances collected in a standard benchmark set. The method could then be further generalized in two steps. First, in a collaboration with Roberto Castañeda Lozano, additional techniques could be incorporated into the approach in order to handle multiple address registers. Fortunately, the methods could then even be further extended to as well deal with more flexible addressing capabilities. In this way, the thesis at hand does not only answer the question how large the address computation overhead can be when using heuristics, but as well presents first results that allow to analyze the impact of the mentioned increased addressing capabilities on the runtime performance and size of real-world programs.

Acknowledgments

This thesis is the outcome of more than five years of research in a wonderful atmosphere thanks to some persons I cannot resist to mention at this point. First of all, I thank Michael Jünger who trusted in a young computer scientist with only a basic background in math, giving him the chance to learn about optimization. I thank you especially for the freedom to develop myself, to learn at my own speed and for giving me a warm welcome irrespective of which concern I brought into your office. I thank you for your positive attitude, for keeping superfluous load off my colleagues' and my back, for treating us as young scientists rather than as assistants.

I thank my colleagues Thomas Lange, Göntje Teuchert, Frank Baumann, Gregor Pardella, Andreas Schmutzer, Daniel Schmidt, Martin Gronemann, Christiane Spisla, Lena Tepaße, Käte Zimmer, Dustin Feld and Francesco Mambelli for fruitful discussions, nice insights, private conversations, pleasant journeys, a lot of fun, and always enough reasons to drink coffee or beer. For the same reasons, I thank Michael Belling, and of course for his support when it comes to the retrieval of hardly accessible literature. I thank Thomas and Göntje for all the administrative support that often went beyond the necessary scope. With all of you there were a lot of moments where you were more than just colleagues! I thank Roberto Castañeda Lozano for a productive collaboration, for giving me some insights into the world of constraint programming and for supporting me at my visit in Sweden. I thank Danny van Dyk for giving me a kickstart into elegant C++ when I was a computer science student. Also I thank him, Dirk Ribbrock, and Markus Geveler for being friends and for keeping up our relationships after finishing our studies. I thank Petra Mutzel and Carsten Gutwenger for their support at the very beginning of my interest and work in research. Doubtless, I thank my family and friends for their constant support during the last years. Finally, I once more thank Lena Tepaße, Roberto Castañeda Lozano, and Daniel Schmidt for their proofreading of this thesis.

Contents

Introduction

This thesis deals with the optimal solution of two optimization problems, *instruction scheduling* and *offset assignment*, that arise in the machine code generation phase of compilers for various processor architectures.

At a high abstraction level, a compiler can be considered to consist of two main components that are respectively called *frontend* and *backend* of the compiler (cf. Fig. 1). Roughly speaking, the frontend takes the input source code, analyzes the control and data flow of the associated program and transforms it into some form of internal representation. The backend then uses this internal representation in order to map the respective operations to machine code instructions of the target architecture and applies several optimizations to it. The two problems addressed in this thesis belong to the machine-dependent code optimizations in the backend part of a compiler. Apart from that, both problems are rather unrelated to each other.

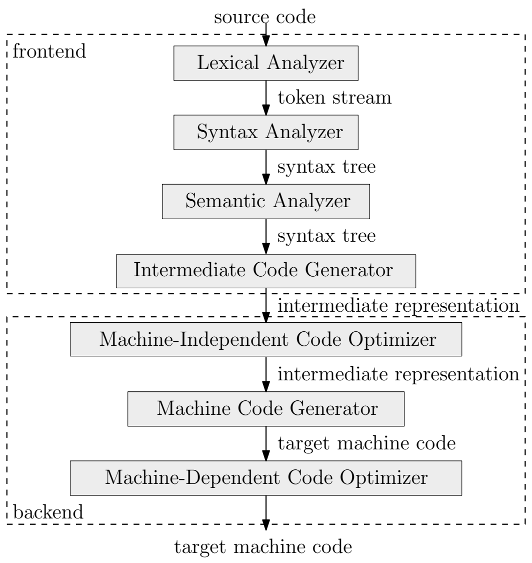


Figure 1: A schematic and highly abstracted illustration of the organization of a compiler. It is a close adaptation of an image from [ALSU86].

An instruction scheduler is implemented into almost any compiler irrespective of the target processor architecture. It is responsible for deriving a processing order of the machine code instructions generated in the *instruction selection* phase (and, depending on the implementation, *register allocation* phase) preceding the instruction scheduling step within the backend. Almost all modern processors are *pipelined*. They partition the execution of an instruction into several *stages* such that a new instruction can enter the first stage while preceding ones are still being processed by the other stages. However, the ideal flow of instructions through the pipeline is harmed by dependencies between instructions on the one hand, and by the varying latencies caused by the processor's functional units and the transmission of data via buses, registers and memory on the other. Functional units realizing more complex instructions need more time in order to make their results available in logic than simpler ones. Some particular operations may even be hard to integrate into the pipeline at all and then need to be synchronized with the flow of pipelined instructions in the case of data dependencies. Further, direct accesses to the main memory may be slower than instructions operating on register values only. A too early insertion of an instruction into the pipeline such that its operands are not yet present in logic when they are needed can be detected by *pipeline interlocks* [HP07]. The pipeline is then stalled for the respective necessary number of clock cycles, effectively prolonging the total execution time. The objective of an instruction scheduler is to avoid such delays caused by pipeline stalls as well as possible. Taking the latencies between dependent instructions into account, an instruction scheduler may reorder the instructions as long as this preserves the semantics of the respective program, i.e., does not violate any data dependencies invoked by the respective computations [ALSU86]. The instruction scheduling phase is therefore a critical step w.r.t. the later runtime performance of a program, especially for *in-order* processors that exactly adhere to the schedule provided by the compiler. Even if a processor is capable of processing instructions *out-of-order*, the runtime performance of a program can be significantly improved by a good or even optimal instruction scheduler.

Offset assignment deals with the placement of variables in memory and with the optimization of accesses to these variables employing the available *address registers* and specialized addressing instructions provided by the processor. Unlike instruction scheduling, the offset assignment problem occurs mainly in compilers for application-specific processor designs with *Harvard architectures* such as, e.g., digital signal processors. Harvard architectures (in contrast to *von Neumann architectures*) strictly separate the storage and transmission of instructions and data. Further, special-purpose processors often come with limitations compared to general-purpose processors because they are designed at more restrictive cost conditions or they are subject to hard constraints w.r.t. power dissipation, operating temperature, or size [Mar06]. The processors we are considering for the offset assignment problem have limited addressing capabilities that can lead to a significant overhead in code size and execution time in that additional instructions are necessary to move between the memory locations to be accessed. The overhead can however be at least partially compensated by a smart application of specialized instructions in combination with an optimized memory layout and address register use. This is a typical situation where the reduced complexity of a processor design leads to an increase of complexity on the software side because the generation of efficient machine code becomes much more difficult. In this context, it is worth mentioning some processor design paradigms that will be referred to especially in the context of scheduling. *Very long* *instruction word* (VLIW) and *explicitly parallel instruction computing* (EPIC) processors adhere to the idea of moving complexity from hardware to software in an extreme fashion. These design classes rely on compilers to specify complex bundles of instructions to be executed in parallel. On the other hand, the reduced instruction set (RISC, for *reduced instruction set computer*) typically found in application-specific processor designs may also simplify some particular tasks of a compiler. For example, register files comprising a comparably large number of general-purpose registers, instruction sets that avoid ambiguity, and a small number of instruction formats can simplify other machine code generation tasks like the already mentioned register allocation and instruction selection phases [ALSU86, HP07].

As indicated, optimizations in the machine code generation phase are crucial for a good runtime performance and also for the size of the resulting programs. Many of them, however, have a high computational complexity which is also true for the two problems dealt with in this thesis. Their $\mathcal{NP}$-hardness is the main reason why they are usually solved heuristically in practice, even though this may result in significant performance penalties. On the one hand, it is true that most computer scientists and mathematicians believe that $\mathcal{P} \neq \mathcal{NP}$ and hence no efficient exact algorithms can be expected for this class of problems. On the other hand, it has been demonstrated several times that the careful adaptation of mathematical optimization techniques to particular problems can permit to compute optimal solutions for instance sizes of practical interest in reasonable time. While it is certainly true that the abstractions used in some 'research problems' can be hard to map to reality and exact approaches are frequently not in a state to be effortlessly used in production compilers, it is sometimes also a matter of missing acceptance if these methods do not find their way to practical use. Nonetheless, the design of exact approaches to compiler optimization problems is being explored by several research groups around the world, not only because optimal solutions are essential in evaluating the quality of heuristics, but also because exact techniques are evolving such that optimal solutions become increasingly achievable. In the domain of technical computer science and engineering, exact algorithms may have even another meaning in answering questions like 'To which extent can the exploitation of a particular feature compensate the lack of another?' as is, e.g., interesting w.r.t. the limited addressing capabilities dealt with in the context of the offset assignment problem. Still the exchange of knowledge between experts in hardware-software co-design, compiler developers and experts in mathematical optimization is improvable. In a way, this thesis aims at being a contribution in this sense, solving problems that the author became aware of when specializing in computer architectures with methods learned when studying mathematical optimization. The main difference of this work to most others in this interdisciplinary field is that the addressed problems are not considered to be 'solved' by mathematically modeling them and then passing them as input to a black-box optimization software. Rather, the combinatorial structure of a problem is investigated and taken into account when designing the respective solution techniques. In particular, existing knowledge about the solution of closely related and well-studied optimization problems, that (partially) characterize the feasible solutions of the problems studied, is exploited in order to implement solvers that remain relatively compact and are able to solve problem instances of medium or even large size. These approaches can as well inspire researchers and practitioners interested in (provably) near-optimal solutions or more sophisticated heuristic and approximation algorithms.

This dissertation starts by establishing the mathematical basis for the applied optimization techniques in the first chapter. The second and third chapters deal with the *basic-block instruction scheduling problem* for *single-issue processors*. Basic blocks are program fragments with no side-entrances and -exits, i.e., every instruction of a basic block needs to be executed before the control flow may leave it and enter another basic block. Single-issue processors (ideally) start the execution of exactly one instruction per clock cycle. Chapter 2 introduces the central definitions related to the problem and discusses exact and heuristic solution approaches from the literature. Emphasis is then laid to some more recent preprocessing techniques that permitted to solve large instances to optimality for the first time. In particular, a constraint programming approach that was published in 2006 optimally schedules about $350,000$ basic blocks of up to about $2,500$ instructions. It was reasonable to suspect that the preprocessing methods can have a positive impact on other solution approaches as well. The last attempt to tackle the problem using integer programming was however not fully equipped with these techniques, and shown not to be competitive to the constraint programming method on the mentioned large benchmark set. Further, the previous integer programming models were almost all based on so-called time-indexed formulations. This was the motivation for a completely different and novel integer programming approach that takes the mentioned preprocessing techniques into account. This approach is the subject of Chapter 3. It is based on *linear ordering variables* and leads to a new characterization of the feasible solutions to the problem. After presenting the models and additional valid inequalities, the corresponding branch-and-cut solver implementation is evaluated using the same test instances that were used for the constraint programming approach. It is capable to solve all but about 150 of the instances to optimality in less than a second of CPU time. There were only eleven instances where it timed out after ten minutes (while only one timed out with the constraint programming solver). The fourth and fifth chapters addressing the offset assignment problem are built up in a similar way. First, Chapter 4 introduces and motivates several variants of the problem, discusses existing approaches as well as known characterizations of the respective sets of feasible solutions, and also some newly developed extensions to these characterizations. Chapter 5 then presents novel integer programming models, starting with the simplest problem variant in which the processor provides only a single address register and very restricted possibilities to avoid address computation overhead. The models are then subsequently extended to deal with multiple address registers and with more flexible addressing capabilities. Finally, the corresponding branch-and-cut solver implementations are evaluated and shown to yield optimal solutions to the majority of about $3,000$ real-world instances collected in a standard benchmark set. The results do not only reveal the overhead incurred when using heuristics, but also allow for a first estimation of the impact of the mentioned addressing capabilities on the total address computation overhead.

Chapter 1

Preliminaries

This chapter covers the mathematical basics of the problems and solution methodologies discussed and developed in this thesis. The presentation is kept as complete and self-contained as necessary but also as restrictive as possible since it is impossible to investigate these topics as comprehensively as in the vast pertinent literature. The reader interested in a deeper discussion is kindly referred to the textbooks and articles cited in the respective sections. Furthermore, familiarity with the basic concepts of computational complexity, in particular the theory of $\mathcal{NP}$-completeness, is assumed.

1.1 Combinatorial Optimization

The compiler optimization problems considered in this thesis are particular instances of a general class of *combinatorial optimization problems*. While several differing definitions of combinatorial optimization problems can be found in the literature, we will identify these problems using the common characteristic that their solutions can be expressed as subsets F of a finite ground set $E = \{e_1, e_2, \ldots, e_n\}$ [NW88]. While we call each subset $F \subseteq E$ a 'solution', we are particularly interested in the set of of *feasible solutions* $\mathcal{F} = \{F \mid F \subseteq E,\ F \text{ feasible}\}$. Further given an objective function $c : \mathcal{F} \to \mathbb{R}$ such that $c(F)$ denotes the objective function value of each feasible solution $F \in \mathcal{F}$, a combinatorial optimization problem Q can be stated as $Q = (E, \mathcal{F}, c) = \max\{c(F) \mid F \in \mathcal{F}\}$. This definition naturally covers also minimization problems that can be obtained by simply negating the coefficients of the objective function. As soon as $\mathcal{F}$ is nonempty, it is clear that the optimum objective function value will be attained by at least one of the feasible solutions since $\mathcal{F}$ is a finite set [KV12]. Otherwise, we will say that the problem is *infeasible*.

Due to its finite nature, a combinatorial optimization problem $Q = (E, \mathcal{F}, c)$ can in principle be solved by inspection of all possible combinations of elements from E, provided that algorithms to evaluate the feasibility of a solution $F \subseteq E$ and the objective function value of a feasible solution $F \in \mathcal{F}$ are available. However, since the number of solutions to inspect is exponential in $|E|$, such an enumerative approach is intractable for larger problem instances. Even in the frequent case that all the feasible solutions to Q can be constructed in a simple fashion, an enumerative approach may be impractical since also the set $\mathcal{F}$ may be of exponential size. Taking the *simple offset assignment problem*, that is addressed in the fourth and fifth chapters of this thesis, as an example, one can consider E to correspond to a set of n program variables and $\mathcal{F}$ to consist of all linear sequences that can be constructed from them. While the construction of each linear sequence is easy, there are $\frac{n!}{2}$ of them to evaluate (a sequence and its reverse sequence have the same objective function value). It is however a key observation that an exponential number of feasible solutions does not necessarily mean that the corresponding problem is not polynomial-time solvable. Regardless whether we can expect a polynomial-time algorithm or not, we are interested in approaches that try to avoid the necessity to explicitly consider the entire search space while determining an optimum solution.

In this thesis, we will mainly deal with the special case of *linear combinatorial optimization problems* where the objective function can be expressed as a linear function $c : E \to \mathbb{R}$. Typically, such an objective function assigns *costs* or *weights* to each of the elements in E and the value of a solution $F \subseteq E$ is then simply given by $c(F) = \sum_{f \in F} c(f)$. Several problems with varying computational complexity fall into this category. On the one hand, there are instances like the *shortest path problem* or the *maximum spanning tree problem* for which polynomial-time combinatorial algorithms exist [CCPS98]. On the other hand, there are (strongly) $\mathcal{NP}$-hard problems like the famous *traveling salesman problem*, the *linear ordering problem* or the instruction scheduling and offset assignment problems that are all considered in this thesis. For these, there are no polynomial-time algorithms unless $\mathcal{P} = \mathcal{NP}$.

1.2 (Integer) Linear Programming

Combinatorial optimization problems as just defined usually admit a characterization of their feasible solutions by linear constraints (equations and inequalities) on a number of integral variables and may then be formulated as an *integer (linear) program* (IP)

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j & & \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq \; b_i & & \text{for all } i \in \{1, \dots, m\} \\ & \qquad x_j \geq \; 0 \text{ and integer} & & \text{for all } j \in \{1, \dots, n\} \end{aligned}$$

or $\max\{c^T x \mid Ax \leq b, x \geq 0 \text{ and integer}\}$ for short [Iba76, Wol98].

Here, A is an $m \times n$ matrix and b is an m-dimensional column vector, c^T is an n-dimensional row vector and x is an n-dimensional column vector of unknowns. Since one usually formulates integer programs in order to solve them using a computer, it is reasonable to assume that all the input data is rational, i.e., $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, and $c \in \mathbb{Q}^n$. The coefficient matrix A (left hand sides) and the vector of right hand sides b together describe the linear constraints that need to be satisfied by (integral) solutions x, and c imposes the linear objective function on the components of x. As for combinatorial optimization problems, assuming a maximization objective is without loss of generality since a minimization problem can again be obtained by negating the objective function coefficients. The above definition naturally covers also equations

$$\sum_{j=1}^{n} a_{ij} x_j = b_i$$

since these can be represented by the two inequalities

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ and } \sum_{j=1}^{n} -a_{ij} x_j \leq -b_i.$$

Similarly, the *trivial inequalities* $x_j \geq 0$ for all $j \in \{1, \dots, n\}$ do not impose a true restriction since a general variable x'_j can be modeled by constraining it to be the difference of two nonnegative variables, i.e., $y - z = x'_j$ with $y, z \geq 0$.

Solving integer programs is $\mathcal{NP}$-hard in general [NW88] and the special case of $\{0,1\}$-IPs where each variable x_j is restricted to be either zero or one is among Karp's compilation of 21 $\mathcal{NP}$-hard problems [Kar72]. If the integrality restriction is imposed only on some of the variables, then the formulation is called a *mixed-integer (linear) program* (MIP) and solving it remains, in general, $\mathcal{NP}$-hard [NW88]. Finally, if there is no integrality restriction at all, then we are given a *linear program* (LP) that can be solved in polynomial time [Kha79, GL81, Kar84].

Throughout this thesis, we will use the following terminology w.r.t. (integer) linear programs. A vector $x^* \in \mathbb{R}^n$, $x \geq 0$, with at least one component $x^*_i \notin \mathbb{N}_0$ will be referred to as being *fractional.* If $Ax^* \leq b$ holds, then x^* will be called a *fractional solution.* For any vector $x^* \in \mathbb{R}^n$, irrespective whether fractional or integral, the inequality $\sum_{j=1}^n a_{ij}x_j \leq b_i$ is said to be *satisfied* (by x^*), if $\sum_{j=1}^n a_{ij}x_j^* \leq b_i$. Further, we call the inequality *binding* (at x^*), if it holds that $\sum_{j=1}^n a_{ij}x_j^* = b_i$. Finally, we say that the inequality is *violated* (by x^*) if $\sum_{j=1}^n a_{ij}x_j^* > b_i$.

1.3 Relaxations

Given some optimization problem Q, a *relaxation* R of Q is another optimization problem that has a potentially larger solution space than Q while every solution to Q is feasible for R as well. Hence, for a maximization problem, if z^* is the value of an optimum solution to Q, then an optimum solution z_R^* to R must have an objective function value $z_R^* \geq z^*$. In other words, z_R^* provides an upper bound on the value z^* of interest. This is useful, e.g., to evaluate the quality of a known feasible solution to problem Q. In particular, if z_R^* coincides with the objective function value of the known solution, then it is proven to be optimal for Q. A relaxation can therefore help to solve the original problem to optimality in an indirect way, for instance by proving a solution to be optimal that has been obtained heuristically.

In the finite context of combinatorial optimization problems, we can describe relaxations as follows. Let $Q = (E, \mathcal{F}, c)$ be the original problem and let $z^* = \max\{c(F) \mid F \in \mathcal{F}\}$ be its optimal objective function value. Then a problem $R = (E, \mathcal{F}', c)$ is a relaxation of Q if $\mathcal{F} \subseteq \mathcal{F}'$ and $z^* \leq \max\{c(F) \mid F \in \mathcal{F}'\}$ [NW88].

Another form of relaxation that we will frequently deal with is a linear program that is derived from an IP (or a MIP) by neglecting any integrality restriction on the variables x_j, $j \in \{1, \ldots, n\}$. It is called the *linear programming relaxation* of the respective program. A linear programming relaxation is a *continuous* relaxation since the set of integer feasible solutions is augmented by all solutions $x \in \mathbb{R}^n$, $x \geq 0$, with fractional components satisfying the inequality system $Ax \leq b$. Moreover, we will see in Sect. 1.4.3 that even the solution sets associated to integer programs need not necessarily be finite. Regardless of that, a linear programming relaxation is a true relaxation since we have that

$$z_{LP}^* = \max\{c^T x \mid Ax \leq b, x \geq 0, x \in \mathbb{R}^n\} \geq \max\{c^T x \mid Ax \leq b, x \geq 0, x \in \mathbb{N}_0^n\} = z^*.$$

For a particular optimization problem Q, the maximum quotient z_{LP}^*/z^* that can be attained for any instance of Q is called the *integrality gap* of the LP relaxation.

1.4 Linear Programming and Polyhedral Theory

A considerable part of the practical success of linear and integer linear programming techniques stems from investigations of the solution sets associated to the respective problem formulations. Before we dive into this topic in more detail, we need some basic concepts from linear algebra. The following descriptions and results are based on the presentations by Grötschel and Padberg [GP85], Schrijver [Sch86], and Nemhauser and Wolsey [NW88]. Most of the results are restated without their associated proofs that can also be found in the cited references.

1.4.1 Combinations and Independence of Vectors, Convexity

Let $k, n \in \mathbb{N}$ such that $k \leq n$. Consider k vectors $x^1, \dots, x^k \in \mathbb{R}^n$ and k scalars $\lambda_1, \dots, \lambda_k \in \mathbb{R}$. A vector $x \in \mathbb{R}^n$ that can be written as $x = \lambda_1 x^1 + \dots + \lambda_k x^k$ is called a *linear combination* of the vectors $x^1, \dots, x^k \in \mathbb{R}^n$. If the additional restriction $\sum_{i=1}^k \lambda_i = 1$ holds, then x is called an *affine combination* of these vectors.

Closely related to these definitions are the following notions of *independence*. If the only linear combination satisfying the equation $0 = \lambda_1 x^1 + \dots + \lambda_k x^k$ has $\lambda_i = 0$ for all $i \in \{1, \dots, k\}$, then the vectors are said to be *linearly independent*. In any other case, the vectors $x^1, \dots, x^k \in \mathbb{R}^n$ are called *linearly dependent*. Similarly, we call the vectors $x^1, \dots, x^k \in \mathbb{R}^n$ *affinely independent* if the only solution to $0 = \lambda_1 x^1 + \dots + \lambda_k x^k$, $\sum_{i=1}^k \lambda_i = 0$, has $\lambda_i = 0$ for all $i \in \{1, \dots, k\}$, and *affinely dependent* otherwise. We observe that the vectors $x^1, \dots, x^k$ are linearly (affinely) independent if and only if none of them can be expressed as a linear (affine) combination of (a subset of) the others.

By definition, any set $S \subseteq \mathbb{R}^n$ that contains the zero vector is linearly dependent. The maximum number of linearly independent vectors from $\mathbb{R}^n$ is n, while the maximum number of affinely independent vectors is $n+1$. In particular, if $x^1, \dots, x^k \in \mathbb{R}^n$ are linearly independent, then $0, x^1, \dots, x^k \in \mathbb{R}^n$ are affinely independent.

Besides linear and affine combinations, we consider *conic combinations* where $\lambda_i \geq 0$ for all $i \in \{1, \dots, k\}$. The vector $x \in \mathbb{R}^n$, $x = \lambda_1 x^1 + \dots + \lambda_k x^k$ is called a *convex combination* of the vectors $x^1, \dots, x^k$ if it is both, an affine and a conic combination. Moreover, we call a set $S \subseteq \mathbb{R}^n$ *convex*, if any convex combination of any two points $x, y \in S$, $z = \lambda x + (1 - \lambda) y$, $\lambda \in [0, 1]$, is again contained in S.

Provided with a set of vectors $x^1, \dots, x^k \in \mathbb{R}^n$, we are particularly interested in the set of points that can be represented by the different forms of combinations of these vectors. The set of all linear combinations of the vectors $x^1, \dots, x^k$,

$$\operatorname{lin}(\{x^1, \dots, x^k\}) := \left\{ \sum_{i=1}^k \lambda_i x^i \mid \lambda \in \mathbb{R}^k \right\},$$

is called their *linear hull*. Especially, if S is a set of n linearly independent vectors in $\mathbb{R}^n$, then $\operatorname{lin}(S) = \mathbb{R}^n$ and S is a *basis* of $\mathbb{R}^n$.

The set of all affine combinations of the vectors $x^1, \dots, x^k$,

$$\operatorname{aff}(\{x^1, \dots, x^k\}) := \left\{ \sum_{i=1}^k \lambda_i x^i \mid \lambda \in \mathbb{R}^k,\ \sum_{i=1}^k \lambda_i = 1 \right\},$$

is called their *affine hull*.

Analogously, the set of all conic combinations of the vectors $x^1, \dots, x^k$,

$$\operatorname{cone}(\{x^1, \dots, x^k\}) := \left\{ \sum_{i=1}^k \lambda_i x^i \mid \lambda \in \mathbb{R}^k,\ \lambda_i \geq 0 \right\},$$

forms a *cone* or a *conic subspace* in $\mathbb{R}^n$.

We will also need the set of all convex combinations of the vectors $x^1, \ldots, x^k$,

$$\operatorname{conv}(\{x^1, \ldots, x^k\}) := \left\{ \sum_{i=1}^{k} \lambda_i x^i \mid \lambda \in \mathbb{R}^k,\ \lambda_i \geq 0 \text{ and } \sum_{i=1}^{k} \lambda_i = 1 \right\},$$

that is called their *convex hull*.

1.4.2 Rank and Dimension

The *rank* of a set $S \subseteq \mathbb{R}^n$, $\operatorname{rank}(S)$, is the maximum number of linearly independent vectors contained in S. Similarly, the *affine rank* of S, $\operatorname{arank}(S)$, is the maximum number of affinely independent vectors contained in S. In particular, if the zero vector can be expressed as an affine combination of the vectors in S ($0 \in \operatorname{aff}(S)$), then $\operatorname{arank}(S) = \operatorname{rank}(S) + 1$, and $\operatorname{arank}(S) = \operatorname{rank}(S)$ otherwise. The *dimension* $\dim(S)$ of S is defined as $\dim(S) = \operatorname{arank}(S) - 1$. In case that $\dim(S) = n$, then the set S is called *full-dimensional*.

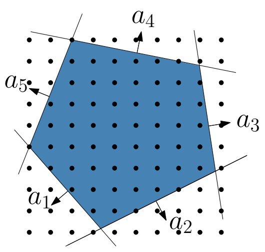
Let $A \in \mathbb{R}^{m \times n}$ be a matrix. The rank of A is equally given by the maximum number of linearly independent rows or columns of A. If A defines the left hand sides of inequalities or equations, we will sometimes say 'linear independent inequalities (equations)' and mean the linear independence of the vectors given by their associated matrix rows. The linear equation system $Ax = b$ has at least one solution $x \in \mathbb{R}^n$ if $\operatorname{rank}(A) = \operatorname{rank}([A\ b])$ where $[A\ b]$ is the *augmented matrix* that results when interpreting b as a further column appended to A. In other words, in order for $Ax = b$ to have a solution, the vector b must be expressible as a linear combination of the columns of A. In particular, $Ax = b$ has a unique solution x^* if $\operatorname{rank}(A) = \operatorname{rank}([A\ b]) = n$.

1.4.3 Polyhedra

The following definition of a *polyhedron* combines those given in [GP85] and [NW88].

Definition 1.4.1. (Polyhedron, Polytope [GP85, NW88]). *A set $P \subseteq \mathbb{R}^n$ is called a polyhedron, if there exists an $m \in \mathbb{Z}_{\geq 0}$, a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$ such that $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$. A polyhedron P is also called a polytope if it is bounded, i.e., if there exists a scalar $w \in \mathbb{R}$, $w \geq 0$, such that $P \subseteq \{x \in \mathbb{R}^n \mid -w \leq x_i \leq w \text{ for all } i \in \{1, \ldots, n\}\}$.*

Comparing this definition with the one given for linear programs in Sect. 1.4, it can be seen directly that the solution set described by the inequalities of an LP is a polyhedron and that the solution set of the associated IP consists of the integral points contained in it. This is depicted in the left picture of Fig. 1.1 for the case of a two-dimensional polytope. Classical results by Minkowski [Min96] and Weyl [Wey35] state that any polyhedron has a second characterization besides the one by linear inequalities. If a polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ is not bounded, then there are vectors $r \in \mathbb{R}^n$ such that $x + \lambda r \in P$ for all $x \in P$ and $\lambda \geq 1$. The vectors r represent an *unbounded direction* of P and are also called *rays* of P. Moreover, the finite set



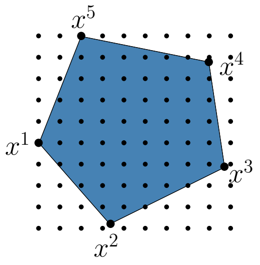


Figure 1.1: A two-dimensional polytope defined by inequalities $a_i^T x \leq b_i$ (left) and as the convex hull of extreme points $x^1, \ldots, x^5 \in \mathbb{R}^2$ (right).

$R = \{r^1, \ldots, r^l\}$ of P's rays that cannot be expressed as a convex combination of any other rays are called P's *extreme rays*. The mentioned classical results state that P can then be expressed by the Minkowski sum of the convex sets given by all convex combinations of its *extreme points* $\{x^1, \ldots, x^k\}$ and all conic combinations of its extreme rays $\{r^1, \ldots, r^l\}$, i.e., $P = \text{conv}\{x^1, \ldots, x^k\} + \text{cone}\{r^1, \ldots, r^l\}$ (cf. Fig. 1.2). Here, the Minkowski sum of two sets $X, Y \subseteq \mathbb{R}^n$ is defined as $X + Y = \{x + y \mid x \in X, y \in Y\}$. If P is a polytope, the conic part is not needed and P can be expressed as the convex hull of its extreme points only, as is depicted in the right picture of Fig. 1.1. While it is possible to transform a polyhedral description by linear inequalities into a description by extreme points (and extreme rays), this is usually an expensive operation. Any extreme point of $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ is given by the intersection of n linearly independent inequalities. Consequently, if the matrix A has m rows, then P can have up to $\binom{m}{n}$ extreme points. Hence, it is possible that the description by extreme points has a size that is exponential in n. Moreover, the number m of inequalities necessary to completely describe P can itself be already exponential in n.

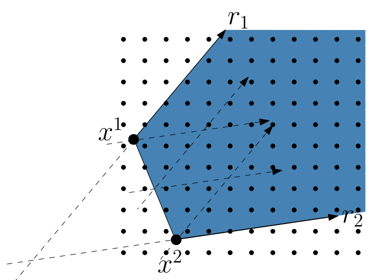


Figure 1.2: A polyhedron defined by the extreme points $x^1, x^2 \in \mathbb{R}^2$ and the extreme rays $r_1, r_2 \in \mathbb{R}^2$. The dashed lines indicate how the Minkowski sum translates the cone spanned by r_1 and r_2 onto $\text{conv}\{x^1, x^2\}$, i.e., the line between x^1 and x^2.

For many polyhedra that are associated with combinatorial optimization problems, no complete description of their feasible region by linear inequalities is known. If it is known though, then the corresponding problem can be solved as a (huge) linear program. In case that the number of the inequalities is polynomial in the input size of the problem, the optimization could then be carried out in polynomial time.

However, usually only a problem description is known that defines a polyhedron whose integral points correspond to the set of feasible solutions, but the inequalities do not define the convex hull of these points. In this case, where a problem Q is modeled as an integer program $\max \{c^T x \mid Ax \leq b, x \geq 0, x \in \mathbb{N}_0^n\}$, i.e., the variables are explicitly constrained to be integer without expressing this in terms of inequalities, often a comparably small number of inequalities suffices in order to characterize the feasible solutions to Q. Still 'comparably small' may mean that the number of necessary constraints is exponential in n. Compare the two polytopes in the left image of Fig. 1.3. Both describe the same set of integer feasible points. However the larger one includes some more fractional points that are exactly cut off by the additional inequalities of the smaller one. Every extreme point of the inner polytope corresponds to an integral feasible solution and lies in the intersection of exactly two inequalities. It can further be observed that the optimal solutions of the inner and outer polytopes do not coincide w.r.t. the example objective function shown in the right picture of Fig. 1.3 that focuses on the upper right corner of the left one.

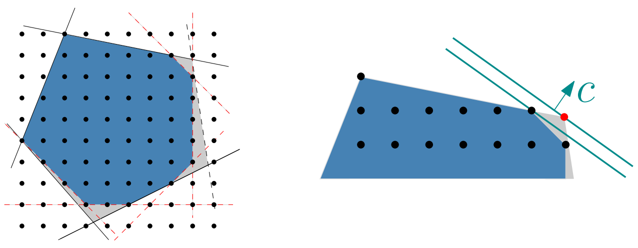


Figure 1.3: Two polytopes with the same set of integral solutions. Only the inner one completely describes their convex hull by linear inequalities. The right image shows an objective function leading to different optima w.r.t. the two polytopes.

1.4.4 Hyperplanes, Faces and Facets

Let $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ be a *polyhedron*. Consider the i-th inequality $a^T x \leq b_i$ of P. If $a \neq 0$, the associated set $S = \{x \in \mathbb{R}^n \mid a^T x \leq b_i\}$ is called a *halfspace* since it partitions the $\mathbb{R}^n$ into the sets of points $\{x \in \mathbb{R}^n \mid a^T x \leq b_i\}$ (that satisfy the inequality) and $\{x \in \mathbb{R}^n \mid a^T x > b_i\}$ (that violate the inequality). In particular, provided that the row vectors of A are all nonzero vectors, they define halfspaces and therefore any polyhedron $P \subsetneq \mathbb{R}^n$ is the intersection of finitely many halfspaces. The boundary $H = \{x \in \mathbb{R}^n \mid a^T x = b_i\}$ of a halfspace is called a *hyperplane*.

Of special interest are inequalities whose associated hyperplanes provide a good characterization of the solutions to a certain problem. An inequality $a^T x \leq b_i$ is called *valid* w.r.t. to a polyhedron P if $P \subseteq \{x \in \mathbb{R}^n \mid a^T x \leq b_i\}$. The associated hyperplane $H = \{x \in \mathbb{R}^n \mid a^T x = b_i\}$ is called a *supporting hyperplane* if $P \cap H$ is nonempty or, equivalently, if $\max \{a^T x \mid x \in P\} = b_i$. The intersection of P with a (supporting) hyperplane H is called a *face* of P. We then say that the face F is defined (or induced) by the inequality $a^T x \leq b_i$ associated to the hyperplane H.

A face F is a *proper face* of P if $F \neq P$ and a *nontrivial face* of P if F is proper and nonempty. Of central interest is the dimension $\dim(F)$ of a face F. It can be shown that if F is a proper face of the polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ with A being an $m \times n$ matrix, then there exists an index set $R \subseteq \{1, \ldots, m\}$ such that $F = \{x \in P \mid \sum_{j=1}^{n} a_{rj} x_j = b_r \text{ for all } r \in R\}$ [GP85]. Let A_R and b_R be the respective rows of A and b. Then the dimension $dim(F)$ of F is given by $dim(F) = n - \text{rank}([A_R\ b_R])$ [Sch86]. In particular, $0 \leq \dim(F) \leq \dim(P) - 1$. We already came across the zero-dimensional faces of a polyhedron P, called *vertices*, that are exactly its extreme points. By the above relation, for any vertex $x^* \in P$ there exists a submatrix A_R of A consisting of n linearly independent rows such that $\text{rank}([A_R\ b_R]) = n$ and $x^* = A_R^{-1} b_R$. The one-dimensional faces of P are referred to as *edges*. Moreover, the faces of dimension $\dim(P) - 1$ are called *facets* and the inequalities corresponding to their supporting hyperplanes are called *facet-inducing.* Facet-inducing inequalities are particularly important due to the following theorem that classifies the role of facets in terms of minimal descriptions of polyhedra.

Theorem 1.4.1. ([NW88]). *Any polyhedron P has a unique (up to scalar multiplication) minimal representation by a finite set of linear equations and inequalities. In particular, for each facet F_i of P, the minimal representation contains an inequality $a^T x \leq b_i$ (unique up to scalar multiplication) representing F_i.*

1.4.5 Basic Solutions and the Simplex Algorithm

We want to elaborate on the connection between polyhedra and linear programs in some more detail. Let us recall from Sect. 1.4.4 that, for any vertex x^* of a polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$, there exists a submatrix A_R consisting of n linearly independent rows of A so that $\text{rank}([A_R\ b_R]) = n$ and $x^* = A_R^{-1} b_R$ is the unique solution to the system $A_R x = b_R$. Geometrically, x^* lies in the intersection of the hyperplanes associated to the n linearly independent rows of A. In the general discussion of polyhedra in Sect. 1.4.3 and Sect. 1.4.4, we did not assume the vectors satisfying the system of inequalities $Ax \leq b$ to be nonnegative but required the $m \times n$ matrix A to have rank n which ensures that the associated polyhedron has at least one vertex. Now, we reinstall the nonnegativity property and consider polyhedra of the form $P = \{x \in \mathbb{R}^n \mid Ax \leq b,\ x \geq 0\}$ corresponding to the feasible regions of linear programs as defined in Sect. 1.2. In this case, we can sacrifice the rank assumption on A because the trivial inequalities, $x_j \geq 0$ for all $j \in \{1, \ldots, n\}$, are clearly linearly independent. So if $rank(A) < n$, then P has vertices that lie on some of the axes. For each such vertex x^*, we may use the corresponding trivial inequalities to build a matrix A_R such that $\text{rank}([A_R\ b_R]) = n$. Since the row and column vectors of A_R form a *basis* of $\mathbb{R}^n$, a vertex solution x^* that is feasible w.r.t. all the inequalities $Ax \leq b$ is also called a *basic feasible solution.* In the special case that the feasible region P is empty, we call the associated linear program *infeasible.* Further, if there is an objective function $c^T x$ such that $\max\{c^T x \mid x \in \mathbb{R}^n,\ Ax \leq b,\ x \geq 0\} = \infty$, then the associated linear program is *unbounded.* If a linear program is neither infeasible nor unbounded then there is an optimum solution that will be attained and the following theorem provides us with a useful characterization of LP optima.

Theorem 1.4.2. ([NW88]). *Let $P = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$ be a nonempty polyhedron and let $\max\{c^T x \mid x \in \mathbb{R}^n, Ax \leq b, x \geq 0\}$ be an objective function that does not correspond to an unbounded direction of P. Then there exists a vertex $x^* \in P$ such that $c^T x^* = \max\{c^T x \mid x \in \mathbb{R}^n, Ax \leq b, x \geq 0\}$.*

Theorem 1.4.2 gives rise to the well-known *simplex algorithm* to solve linear programs. The simplex algorithm finds the vertices of P indirectly by considering the possible combinations of (at least) n inequalities (those from A plus the trivial ones) to be satisfied with equality. A first feasible basic solution can be found by solving an auxiliary problem with a modified objective function (and potentially some additional column exchange operations). This is usually referred to as *phase one* of the simplex algorithm. Assuming a maximization problem, the main routine (*phase two*) of the algorithm starts from a basic feasible solution and then iteratively moves to an adjacent basic feasible solution (another vertex) with a nondecreasing objective function value as long as this is possible. By Theorem 1.4.2, it must eventually find an optimum solution to the linear program. The motion from one basic solution to an adjacent one is carried out by exchanging so-called basic and non-basic variables (within the simplex algorithm, there are more than n variables due to the addition of slack variables in order to turn inequalities into equations). In general, there can be several candidate variables to make (non-)basic at each iteration and the subroutine to decide which one to choose is known as the *pivoting rule*. There is no proof known for any particular pivoting rule that bounds the number of iterations necessary to arrive at an optimum vertex by a polynomial in the size of the input data. However, and even though provably polynomial-time LP methods exist, the simplex algorithm is still frequently used in practice. Some important reasons for this fact are that (i) the number of iterations observed in practice is often small, (ii) solution methods for linear equation systems can be implemented very efficiently, and (iii) particular variants of the simplex algorithms can be easily 'warmstarted' after adding variables or constraints to the linear program [Sch86]. This will be particularly helpful for the techniques discussed in Sect. 1.5. Nonetheless, for various pivoting rules used in practice, worst case instances with an exponential number of vertices traversed have been reported (see, e.g., [Sch86]).

1.4.6 Cutting Planes and their Separation

1.4.6.1 Cutting Planes

Suppose we want to solve the linear program $\max\{c^T x \mid x \in \mathbb{R}^n, Ax \leq b, x \geq 0\}$ and assume that the number m of constraints (of A's rows) is large. As we already stated, m may be even a number that is exponential in n for some particular integer programming formulations of combinatorial optimization problems. So consider the case where we relax the linear program by neglecting some of its inequalities. In principle, it suffices to keep enough inequalities such that the associated polyhedron is not unbounded w.r.t. the objective function $c^T x$. Let A^1 be a matrix that results from removing some of A's rows, let b^1 be the right hand sides corresponding to the rows in A^1 and suppose that we solve the LP $\max\{c^T x \mid x \in \mathbb{R}^n, A^1 x \leq b^1, x \geq 0\}$.

Let x^* be a solution to this relaxed linear program. Then two cases may occur: Either, it also holds that $Ax^* \leq b$, i.e., the solution x^* satisfies the neglected inequalities without having this enforced explicitly. Or, there exists at least one inequality $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ such that $\sum_{j=1}^{n} a_{ij}x_j^* > b_i$. By subsequently adding the inequality to the relaxed linear program, we know that x^* cannot be a solution anymore if we resolve it. We also say, that the solution x^* is *cut off* or *separated* by the inequality and call it a *cutting plane* or shortly a *cut*. By the interleaved solution of linear programs and addition of cutting planes, we will finally arrive at some solution x^* that is feasible for our original problem, i.e., x^* is an optimum solution to the linear program $\max\{c^Tx \mid x \in \mathbb{R}^n,\ Ax \leq b,\ x \geq 0\}$. When using cutting plane algorithms, we hope to arrive at such a solution without the necessity to add all of the initially neglected inequalities. In general, to avoid that LP relaxations become too large, it is also possible to remove some of the separated inequalities again at a later point of time. A typical strategy is then to remove those inequalities that were satisfied with a large slack in the last LPs, i.e., the term $b_i - \sum_{j=1}^{n} a_{ij}x_j$ is (comparably) large.

1.4.6.2 Separation

The cutting plane approach just indicated gives rise to the so-called *separation problem.* Given an LP solution x^*, find a yet neglected inequality $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ such that $\sum_{j=1}^{n} a_{ij}x_j^* > b_i$ or prove that no such inequality exists. Besides the option to separate necessary inequalities, i.e., those that are needed to characterize the set of integer feasible solutions, one may also consider to separate additional inequalities that are not necessary in this sense but cut off further fractional solutions (cf. Sect. 1.4.3). However, the separation problem associated to a particular class of inequalities may itself be $\mathcal{NP}$-hard. In this case one might consider to design a heuristic separation procedure that may find one or several violated inequalities if they exist but cannot prove that no such inequality exists.

The following famous result is usually attributed to Grötschel, Lovász and Schrijver [GLS81, GLS84]. According to Korte and Vygen [KV12], it has been independently discovered also by Karp and Papadimitriou [KP80, KP82], and Padberg and Rao [PR81]. Here, the theorem is stated in a way that suits for our purposes while the original formulation is even more general.

Theorem 1.4.3. ([GLS81, GLS84]). *An optimization problem (over a bounded polyhedron P) can be solved in polynomial time if and only if its associated separation problem (given P and a point $x^* \in \mathbb{R}^n$, either conclude that $x^* \in P$ or find an inequality $a^Tx \leq b$ such that $a^Tx^* > b$) can be solved in polynomial time.*

It should be mentioned that this theorem has been proven based on the ellipsoid method [GLS81, GLS84] and considers general points $x^* \in \mathbb{R}^n$ (not necessarily vertices), as this will be of importance for a discussion in the following subsection. While each different class of inequalities has its own associated separation problem, Theorem 1.4.3 refers to *all* the separation problems of any nonredundant class of inequalities that define the polyhedron P associated to the feasible solutions of the optimization problem. In particular, if the optimization problem requires integrality of its variables, then the theorem states that the optimization problem is polynomial-time solvable if and only if all inequalities necessary to obtain the integer hull of the polyhedron are separable in polynomial time.

1.4.6.3 General Cuts for Integer Programs: Chvátal-Gomory Cuts

Related to the previous considerations is the question whether we can separate certain classes of inequalities that are valid for each integral solution but violated by any fractional solution of an integer program. A well-known class of inequalities of this kind are the *Chvátal-Gomory cuts* that can be applied to any linear program with integral input data, i.e., a linear program $\max\{c^T x \mid x \in \mathbb{R}^n,\ Ax \le b,\ x \ge 0\}$ with $A \in \mathbb{Z}^{m\times n}$ and $b \in \mathbb{Z}^m$. They can be generated as follows.

We first recognize that any inequality that results as a nonnegative linear combination of some of the inequalities from $Ax \le b$ by choosing a $\lambda \in \mathbb{R}^m$, $\lambda \ge 0$,

$$\sum_{i=1}^{m} \lambda_i \sum_{j=1}^{n} a_{ij} x_j \le \sum_{i=1}^{m} \lambda_i b_i,$$

is a valid inequality to the original linear program. For ease of notation, define $a'_{ij} = \sum_{i=1}^{m} \lambda_i a_{ij}$ and $b'_i = \sum_{i=1}^{m} \lambda_i b_i$, so that we can rewrite the above inequality as

$$\sum_{j=1}^{n} a'_{ij} x_j \le b'_i.$$

Since $x \ge 0$ and $\sum_{j=1}^{n} (a'_{ij} - \lfloor a'_{ij} \rfloor) \ge 0$, any solution satisfying this inequality must also satisfy the inequality

$$\sum_{j=1}^{n} \lfloor a'_{ij} \rfloor x_j \le b'_i.$$

Now, since $\lfloor a'_{ij} \rfloor x_j$ is integer for any $x_j \in \mathbb{Z}$, we can finally round down also the right hand side and obtain the Chvátal-Gomory inequality

$$\sum_{j=1}^{n} \lfloor a'_{ij} \rfloor x_j \le \lfloor b'_i \rfloor.$$

Gomory [Gom58, Gom63] showed that one can derive a violated Chvátal-Gomory inequality from each fractional basic feasible solution. Moreover, Chvátal [Chv73] showed that the integer hull of the polyhedron associated to the LP $\max\{c^T x \mid x \in \mathbb{R}^n,\ Ax \le b,\ x \ge 0\}$ with $A \in \mathbb{Z}^{m\times n}$ and $b \in \mathbb{Z}^m$ can be obtained by adding a finite number of Chvátal-Gomory inequalities. Being more precise, he proved that a finite number of inequalities suffices in order to obtain the polyhedron that results from adding all the Chvátal-Gomory inequalities associated to the original inequality system $Ax \le b$ (called the first *Chvátal-Gomory closure*). The polyhedron obtained by adding all Chvátal-Gomory inequalities corresponding to the first Chvátal-Gomory closure is called the second closure and so on. Chvátal then showed that there exists some i such that the i-th Chvátal-Gomory closure is the integer hull of $Ax \le b$.

At first glance, it looks like we are now stuck in a contradiction. One the one hand, we know that we can compute the integer hull of any polyhedron given by integral input data using Chvátal-Gomory inequalities only, and we are given a polynomial-time separation procedure that constructs such an inequality for each fractional solution. And on the other, we know that the solution of general integer programs is $\mathcal{NP}$-hard. Combined with Theorem 1.4.3 that states equivalence of optimization and separation, this appears to be impossible. However, there is no contradiction, since the theorem is based on the ellipsoid method - and in this method one cannot make assumptions about LP solutions x^* such as requiring them to be a vertex of the respective polyhedron [FL07]. In fact, the general separation problem that, given *any* point $x^* \in \mathbb{R}^n$, decides whether there exists a $\lambda \in \mathbb{R}^m$ such that $\lfloor \lambda^T A \rfloor x^* > \lfloor \lambda^T b \rfloor$, is $\mathcal{NP}$-hard [Eis99]. Moreover, despite the fact that Gomory's separation method can be performed in polynomial time w.r.t. the size of the LP, the number of such Chvátal-Gomory inequalities necessary to reach the integer hull can be exponential in the number of the original inequalities [FL07].

To close this subsection, it is worth mentioning a special case of Chvátal-Gomory inequalities that will be referred to in Chapter 3. These inequalities where all $\lambda_i \in \{0, \frac{1}{2}\}$, $i = 1, \ldots, m$ are called $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts or, for short, just $\{0, \frac{1}{2}\}$-cuts. The separation problem associated to $\{0, \frac{1}{2}\}$-cuts is $\mathcal{NP}$-hard [CF96] as well.

1.4.7 Total Unimodularity

For some particular combinatorial optimization problems, we are in the fortunate situation that there is no need to explicitly enforce the integrality of the variables used in linear programming formulations. This is the case if the associated constraint matrix A has only integral entries and the determinant of each square submatrix of A is either zero, one, or minus one. Such a matrix is called *totally unimodular.* The following variant of a theorem by Hoffman and Kruskal [HK56], that is taken from [CCPS98], provides us with the opportunity to solve the respective problem in polynomial time using an LP while being guaranteed to obtain an integral solution.

Theorem 1.4.4. ([CCPS98]). *Let $A \in \mathbb{Z}^{m \times n}$ be an integral matrix. Then the polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b,\ x \geq 0\}$ has exceptionally integer basic feasible solutions for any $b \in \mathbb{Z}^m$ if and only if A is totally unimodular.*

1.5 Branch-and-Bound and Branch-and-Cut

This section covers some basic concepts of the two solution methodologies named in the title. It deals only with those concepts that were used and implemented for the models presented in this thesis, or at least referred to at some points in this thesis. For the many possible variants, tuning parameters, and extensions to these methods, we refer the interested reader to [EGJR01, Thi95], that served as a basis for the descriptions made here, and to the corresponding textbooks about the solution of general integer programs.

1.5.1 Branch-and-Bound

Branch-and bound is a general enumerative approach to combinatorial optimization problems that resembles a particular instance of *divide and conquer* algorithms. Using the definition of combinatorial optimization problems from Sect. 1.1, *branching* means to decompose a problem $Q = (E, \mathcal{F}, c) = \max\{c(F) \mid F \in \mathcal{F}\}$ into a finite set of smaller subproblems $\{Q_i = (E, \mathcal{F}_i, c)\}$, $i \in \{1, \ldots, k\}$, in such a way that $\bigcup_i^k \mathcal{F}_i = \mathcal{F}$. The subproblems Q_i are derived from Q by adding further constraints to the problem formulation or by restricting the feasible ranges of some of its variables (which is technically the same since any bound on the value of a variable can be expressed as a constraint). The recursive decomposition of a problem into smaller subproblems yields a tree, called the *branch-and-bound tree* which has the original problem as its root. Each (sub-)problem Q_i is represented by a vertex whose adjacent children are the subproblems created from it. Hence, the subtree rooted at Q_i is made up of all the subproblems that are recursively generated from Q_i. In particular, none of these subproblems can contain a solution that is better than the best one to Q_i. The *bounding* character of the approach comes from the idea to maintain and exploit lower and upper bounds on the optimum objective function value (as before, we assume a maximization objective) in the following fashion.

- Any feasible solution found in any subproblem of the branch-and-bound tree yields a lower bound on the optimum objective function value. The best, i.e. maximal, among these is a *global lower bound* and a solution that attains this currently best known objective function value is called an *incumbent solution.* A different opportunity to obtain incumbent solutions and global lower bounds is to use a heuristic or approximation algorithm.

- If an appropriate method is available, a *local upper bound* for each subproblem may be computed (e.g., by relaxing it such that an upper bound can be easily retrieved). If the local upper bound is smaller than or equal to the global lower bound, it is not necessary to further consider the subtree associated to this subproblem in which case we say that it is *pruned* or *fathomed.* The same is true if the subproblem (or its relaxation) turns out to be infeasible.

- The largest among all local upper bounds of nonfathomed subproblems is a *global upper bound* on the optimal objective function value. In particular, if this value is found to be smaller than or equal to the global lower bound, then the current incumbent solution is proven to be optimal.

In the context of integer programming, each subproblem of the branch-and-bound tree is an integer program and the relaxation that is solved there is usually its linear programming relaxation (other opportunities exist). If the LP solution associated to any subproblem is integer (and all necessary constraints are either present or at least not violated), then the subproblem is solved and, potentially, a new incumbent solution and global lower bound is found. If the solution is not integer and the subproblem is not infeasible, then let x_i^* be some fractional variable. The common approach to carry out a branching step is to restrict x_i to be less or equal to $\lfloor x_i \rfloor$ in Q_1 and to greater than or equal to $\lceil x_i \rceil$ in Q_2. Clearly, this procedure does neither add nor remove any integer feasible solutions when both problems are considered to replace the former one. Again, there exist several alternatives and it is also possible to create more than two subproblems. The union of the set of feasible solutions to the created subproblems must however match the feasible solutions of the decomposed one. In general, the routine that specifies how to select a variable to branch on (or the constraints to add), and how subproblems shall be created, is called a *branching rule*. A computationally intensive strategy is to tentatively carry out branching decisions for multiple candidates (e.g., variables) and to solve all the associated subproblem relaxations in order to decide which branching candidate to take ultimately. This strategy is called *strong branching*. Further situations exist in which decisions must be made that may have a large impact on the solution process. For example, whenever there are a number of created but not yet solved (*open*) subproblems, it must be decided which one to process next. A depth-first construction of the branch-and-bound tree emphasizes on a quick generation of feasible solutions. A breadth-first construction promotes the retrieval of useful upper bounds. Several other strategies that try to combine the advantages (and alleviate the disadvantages) of these two extreme strategies exist. For instance, the *best-first* strategy selects a subproblem that has the currently weakest (largest) local upper bound. One the one hand, this subproblem currently offers the best opportunity to comprise a solution with maximum objective function value. And on the other, if this subproblem is the only one with the currently weakest upper bound and this value can be improved by the branching step, then this will also improve the global upper bound.

Another common concept that is applied in combination with LP-based branch-and-bound are *primal heuristics*. Primal heuristics try to construct a feasible solution by exploiting the LP solution obtained at a subproblem. Since the LP solved is a relaxation, is it reasonable to assume that, in general, one may conclude on the importance of some of the variables w.r.t. optimal solutions by inspecting their LP values. For some particular problems, it is then easy to implement some form of a greedy algorithm in order to construct a feasible solution. But of course also more sophisticated variants are possible, yielding a trade-off situation between running time and the quality of solutions. Since linear programs often remain with at least some fractional variables, primal heuristics are often crucial for a quick computation of strong (i.e., large) global lower bounds.

1.5.2 Branch-and-Cut

Branch-and-cut describes the integration of LP-based branch-and-bound with the generation of cutting planes. The solution of *one* linear programming relaxation at each branch-and-bound vertex is replaced by the solution of (potentially) several linear programs that arise from the iterative addition of cutting planes. Since every added cutting plane is a further restriction of the feasible region, one may hope to obtain better local upper bounds at the respective subproblems or to even arrive at an integer feasible solution. Again, several tuning parameters can be considered that can have a larger impact of the solution process. Of importance are especially the parameters that determine (i) the number of iterations where cutting plane generation shall be considered, (ii) a limit on the number of cutting planes generated per iteration, and (iii) which classes of inequalities shall be separated in which order. There is often a trade-off between the invested time for separation and the resulting improvement of the solution process. One may of course add cutting planes until no more further ones of a certain class of inequalities can be found, or install a fixed number of maximum iterations. A more sophisticated idea is to add cutting planes only as long as they have some impact on the objective function value, i.e., lead to better local upper bounds. This is known as a *tailing-off* strategy.

1.6 Standard Linearization Approaches

When formulating combinatorial optimization problems, it is sometimes necessary to model a class of inequalities or an objective function that involves products of variables. Since the inequalities as well as the objective function need to be linear expressions in the variables in order to use linear programming techniques, it may be convenient to apply a linearization to the quadratic (or even higher degree) terms of a model. The following shortly summarizes some standard methods [McC76] to linearize products of variables since they will be applied frequently in this thesis.

We first discuss the linearization of a product of two $\{0,1\}$-variables. Let $x_i, x_j \in \{0,1\}$ be two binary variables and suppose we want to model the product of x_i and x_j. Then this can be modeled by introducing a new variable $y_{ij} \in [0,1]$ and enforcing it to take the value of the product for any integral solution using the inequalities:

$$\begin{aligned} y_{ij} &\leq x_i \\ y_{ij} &\leq x_j \\ y_{ij} &\geq x_i + x_j - 1 \end{aligned}$$

The correctness of this construction is easy to verify. The variable $y_{ij} = x_i x_j$ is equal to one, if and only if x_i and x_j are both one. In this case, the third constraint enforces y_{ij} to attain value one while the first two ones impose no restrictions on y_{ij} (since $y_{ij} \leq 1$ always holds). In the other cases where either x_i or x_j (or both) are zero, the first two inequalities make sure that y_{ij} is also zero while the third one becomes redundant (since $y_{ij} \geq 0$ always holds). It is not necessary to explicitly require integrality of y_{ij} in the problem formulation, since the above inequalities make sure that y_{ij} is integer whenever x_i and x_j are.

Now we move to the case of a product of a $\{0,1\}$- and a general integer variable. Let $x \in \{0,1\}$ be a binary variable and $y \in \mathbb{Z}, y \geq 0$ be a general integer variable. Suppose further, that $y \leq Y$ holds for some upper bound Y. Then a variable $z = xy$ can be constructed in a similar manner to the binary case:

$$\begin{aligned} z &\leq y \\ z &\leq Yx \\ z &\geq y + Yx - Y \end{aligned}$$

This time, we want to achieve that z attains the value of y if x is one, and that it attains value zero if x or y is zero. The latter cases are again handled by the first two constraints. The coefficient Y in the second inequality is necessary to not be too restrictive in the case $x = 1$, since then it must be possible for z to attain any value of y. So if $x = 1$, then the term $Yx - Y$ in the third inequality equals zero, and (together with the first inequality) z is enforced to attain the same value as y. On the other hand, if $x = 0$, then the subtraction of Y in the third inequality makes sure that this constraint is never binding for z independent from the value of y.

1.7 Constraint Programming

Like in integer programming, a *constraint programming model* or *constraint satisfaction problem* characterizes solutions to a problem by formulating constraints over a finite set of variables $X = \{x_1, \ldots, x_n\}$. Each of the variables x_i has an associated *domain* $D(x_i)$ that specifies the possible values x_i may attain. Constraints are expressions on allowed combinations of values for the variables X. A solution to a constraint satisfaction problem is an assignment of values to all the variables such that $x_i \in D(x_i)$ for all $i \in \{1, \ldots, n\}$ and all constraints are satisfied. A central difference between constraint programming (CP) and LP-based integer programming approaches is that integrality is never relaxed in CP solvers. Instead, the optimization process interleaves the application of branching and *propagation* phases. After fixing some variables at certain values, the constraints are used to remove infeasible values from the domains of not yet fixed variables, as we will describe in further detail below. Iterating this procedure leads to either the construction of a feasible solution or the detection that none exists with the currently fixed variable setting. In the latter case, a backtracking in the branch-and-bound tree is carried out. So in contrast to integer programming, constraint programming lays emphasis on feasibility instead of optimality. While it is, in general, possible to formulate any objective function as a constraint and, therefore, to incorporate it into a CP model, it is not necessarily straightforward to direct the search towards optimal solutions. A common way to use constraint programming for optimization is to solve feasibility (or decision) problems associated to distinct objective function values and then to show whether solutions with these values exist or do not exist [Hoo11].

Another difference to integer programming is that the constraints of a CP model may also have a symbolic character or nonlinear nature. An illustrative example for such a constraint, that is also used in the CP models addressed in Chapter 2, is the `alldifferent` constraint. The following explanations are based on the description in [vH01]. The constraint

$$\texttt{alldifferent}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid d_i \in D(x_i), d_i \neq d_j \text{ for } i \neq j\},$$

states that all of the variables $x_1, \ldots, x_n$ have to attain different values. This is equivalent to an expression of the form $x_1 \neq x_2 \neq \cdots \neq x_n$ that is nonlinear. Nonlinear constraints are possible since solutions to a CP model are not computed by optimizing over a polyhedron. Instead, as already indicated, solutions are constructed and the *consistency* of constraints is tested iteratively. This is done by special *filtering* algorithms for each of the constraints. Based on the already assigned values, they try to exploit logical implications in order to reduce the domains on yet undecided variables by removing values that can never appear in any solution to the constraint in the remaining search space. If the domain of a variable is reduced, it is possible that this has implications on other constraints where this variable appears so that their corresponding filtering algorithms are afterwards invoked as well. This process is called *constraint propagation.* If at a certain stage no more inconsistent values are detected, the filtering algorithms have reached some distinct form of consistency. There exist different forms and levels of consistency some of which we want to address in the following. Thereby we assume that the involved variable domains are finite subsets of an ordered base set.

Definition 1.7.1. (Domain consistency [vH01]). *We call a constraint $C(x_1, \dots, x_m)$, $m > 1$, domain consistent if, for all $i \in \{1, \dots, m\}$ and all values $d_i \in D(x_i)$, there exist values $d_j \in D(x_j)$ for all $j \in \{1, \dots, m\} \setminus \{i\}$ such that $(d_1, \dots, d_m)$ satisfies C.*

The meaning of domain consistency is basically that if any variable being part of the constraint is selected and fixed at one of its potential values, then there must be feasible values for all of the other variables present in their domains. This is a strong form of consistency which might be further relaxed in several ways.

Definition 1.7.2. (Range consistency [vH01]). *We call a constraint $C(x_1, \dots, x_m)$, $m > 1$, range consistent if, for all $i \in \{1, \dots, m\}$ and all values $d_i \in D(x_i)$, there exist values $d_j \in [\min D(x_j), \max D(x_j)]$ for all $j \in \{1, \dots, m\} \setminus \{i\}$ such that $(d_1, \dots, d_m)$ satisfies C.*

The intrinsic relaxation of range consistency over domain consistency is that the domains of variables are now treated as they would be dense, i.e., fully populated intervals.

Definition 1.7.3. (Bounds consistency [vH01]). *We call a constraint $C(x_1, \dots, x_m)$, $m > 1$, bounds consistent if, for all $i \in \{1, \dots, m\}$ and all value assignments $d_i \in \{\min D(x_i), \max D(x_i)\}$, there exist values $d_j \in [\min D(x_j), \max D(x_j)]$ for all $j \in \{1, \dots, m\} \setminus \{i\}$ such that $(d_1, \dots, d_m)$ satisfies C.*

To be bounds consistent, a constraint needs to be satisfiable only for each extreme domain value assignment of each of its associated variables, again assuming dense intervals for all of the variables like in the case of range consistency.

Since weaker levels of consistency are typically faster to establish, the different levels allow for a trade-off between the amount of propagation (and therefore, reduction of the remaining search space) and the running time invested at each subproblem of the branch-and-bound tree.

Chapter 2

Instruction Scheduling

This chapter introduces, explains and motivates instruction scheduling which is the main compiler optimization problem dealt with in this thesis. It also provides an overview of existing solution approaches as well as basic and commonly used definitions and notations. After that, several techniques to reduce the search space for optimal solutions to the basic-block instruction scheduling problem are discussed. These include existing as well as newly developed reduction approaches. Further, ideas to improve or better exploit some of the existing methods are presented. As a whole, the chapter prepares for the novel exact basic-block instruction scheduling models presented in the subsequent chapter.

2.1 Motivation

In this chapter, we consider the task to optimally schedule a set of instructions to be executed by a single *pipelined single-issue* processor. A *pipelined* processor partitions the execution of an instruction into several steps called *pipeline stages* [ALSU86]. Typically, a new instruction can be inserted into the pipeline (*issued*) every clock cycle while preceding instructions are still being processed by the other stages. This way, an instruction pipeline yields one form of *instruction-level parallelism.* An ideal flow of instructions through a simple five-stage pipeline for (integer) arithmetical and logical operations in a load-store RISC architecture, as it has been implemented for instance in the MIPS R2000 processor [HP07], is shown in Fig. 2.1.

	Clock cycle							
Instruction	1	2	3	4	5	6	7	8
I_1	IF	ID	EX	MEM	WB			
I_2		IF	ID	EX	MEM	WB		
I_3			IF	ID	EX	MEM	WB	
I_3				IF	ID	EX	MEM	WB
I_4					IF	ID	EX	MEM
I_5						IF	ID	EX

Figure 2.1: Instructions passing through a five-stage instruction pipeline.

In the *instruction fetch* (IF) stage, an instruction is read from the instruction cache using the program counter that is afterwards incremented. The fetched instruction is then *decoded* (ID stage), i.e., the logic involved in executing the instruction is prepared and operands are read from source registers. In case of a branch instruction, the potential branch target address is computed at this point. In the *execution stage* (EX), arithmetical and logical operations are performed or, if the instruction is a load or a store, the effective memory address to be accessed is calculated. For the latter, the actual *memory access* is then carried out in stage MEM. The result of an arithmetical or logical instruction as well as the operand of a load instruction is written back into a register in the *write back* stage (WB).

Pipeline stages of classical RISC processors are designed such that their associated operations can be performed within exactly one clock cycle (*uni-cycle instructions*). Since each instruction needs to pass every pipeline stage, the ideal number of clock cycles needed to complete the execution of an instruction is equal to the number of stages which is also referred to as the *depth* of the pipeline. The 'slowest' stage limits the potential clock frequency. Modern processor designs further subdivide the execution of an instruction which leads to a deeper pipeline, more instruction-level parallelism, and a higher clock frequency [HP07]. In order to achieve additional instruction-level parallelism, many processors are also capable to issue more than one instruction per clock cycle (*single-issue* vs. *multiple-issue* processors) [ALSU86].

Several conflicts or *hazards* may disturb the ideal flow of instructions through the pipeline. One distinguishes three main types of disturbances, namely *data*, *structural* and *control hazards* [HP07]. The main motivation for instruction scheduling are data hazards that reflect the cases where two instructions need to be processed in a particular order to ensure correct results. They are caused by *data dependencies* and, depending on the processor design, by limited capabilities to make results produced by one pipeline stage accessible to another one. One classifies *read after write* (RAW), *write after read* (WAR), and *write after write* (WAW) data dependencies [HP07]. A WAW dependency reflects the case where an instruction I_2 is writing to the same register or memory location that is also written to by an instruction I_1 issued earlier. This can only be a problem if the pipeline is designed such that results are written in more than one stage or multiple pipelines are present. Then, it must be made sure that the write performed by I_2 indeed takes place after the write of I_1. Similarly, in the case of a WAR dependency, I_2 writes to a location that is read by I_1 such that I_1 must read the value before I_2 overwrites it. These two types of dependencies impose orders on instructions but usually do not add any other restrictions. If they are not accompanied by an additional RAW dependency, i.e., they only refer to the same storage locations but not to the stored data, they can potentially be resolved by substituting the assigned registers (register renaming) or by using different or additional (load and store) instructions. At this point, instruction scheduling interferes with other central compiler tasks, namely especially with *register allocation* and sometimes also with *instruction selection* [ALSU86]. Such an alternative resolution is not possible for RAW dependencies which are in the main focus of instruction scheduling. If an instruction I_2 uses the result of an instruction I_1, it must be made sure that the respective operand is available in logic when I_2 will try to access it. Suppose I_1 and I_2 are arithmetical instructions. Without any further modification to the exemplified pipeline, I_2 cannot enter the ID stage before I_1 has completed the WB stage. Hence, I_2 needs to be delayed for three clock cycles after I_1 is issued. However, in case that no other instruction is ready to be executed in the meantime, I_2 can be issued but cannot leave the IF stage for the three cycles as is indicated in Fig. 2.2. As a result the ID, EX, MEM and WB stages cannot perform any useful work in the clock cycles $3-5$, $4-6$, $5-7$ and $6-8$ respectively. Whenever an instruction cannot proceed to the next pipeline stage or no instruction can be issued, the pipeline is said to be *stalled*. Being idle this way can also be seen as executing a ‘no operation’ instruction (*NOP* for short).

	Clock cycle							
Instruction	1	2	3	4	5	6	7	8
I_1	IF	ID	EX	MEM	WB			
I_2		IF	stall	stall	stall	ID	EX	MEM
I_3						IF	ID	EX

Figure 2.2: Pipeline stalls due to data hazards.

Any stall has the effect that only the instructions issued before the one that causes the stall proceed normally while all others are delayed by one clock cycle. The increase of one cycle in the overall execution time is therefore independent from the stage where the stall occurs. Neglecting the precise states of hardware units and considering only the resulting execution time, it is equivalent to insert a NOP before issuing an instruction that would otherwise cause a stall or to stall the pipeline for one cycle after issuing the instruction.

Modern processors mitigate data hazards by *forwarding* (or *bypassing*), i.e., by introducing direct data paths between different stages of the pipeline. For instance, for the exemplary MIPS pipeline, results calculated in the EX stage can be immediately used as an input operand to the successive instructions entering that stage, and the results produced in the MEM stage can be used by the next two successive instructions (entering EX and MEM stages respectively) [HP07]. However, not all data hazards can be resolved like this even if every pair of stages is equipped with forwarding control logic. This is true since it can happen that a result is calculated just in the same or even a later cycle as it is needed by a successively issued instruction [HP07]. For example, if an instruction in the ID stage needs the result of an arithmetical instruction that is just in the EX stage at the same time. Even more, with increasing pipeline depths, more complex instruction types, and additional nonpipelined functional units, different delays have to be respected when one dependent instruction follows another. The actual delays are therefore highly architecture-dependent but known in advance and can hence be taken into account by a compiler. To do this, a *latency* is associated with each dependency between two instructions I_1 and I_2. It specifies the number of clock cycles that need to pass by after issuing I_1 and before I_2 can be issued without causing a pipeline stall. These clock cycles can then be filled with either different instructions or, if unavoidable, with NOPs. This way, the problem to derive a timing of the instructions that preserves the semantics of the program and leads to as few NOPs as possible can be solved by an instruction scheduler during machine code generation.

Another potential reason for pipeline stalls are *structural hazards*. They occur if two instructions cannot execute at the same time since they compete for limited resources of the processor such as special purpose registers, functional units or data paths. This can be a challenging problem especially for the case of multiple-issue and VLIW processors or when instruction scheduling is integrated with register allocation. However, in an extreme setting, structural hazards might even occur for a single functional unit that is responsible for different tasks of two or more pipeline stages. As a simple example, if a processor uses the same data path for loading operands and instructions, every load instruction currently in the MEM stage would interfere with any other instruction that shall be fetched in the IF stage [HP07]. Again, a pipeline stall would be necessary to resolve the conflict. Like in this example, stalls caused by structural hazards are often unavoidable and due to a cost-oriented processor design that avoids the replication of resources that are seldom used in parallel or where the penalty of pipeline stalls is acceptable compared to the cost of additional hardware. In the absence of other dependencies, structural hazards do not impose an order on the two interfering instructions. Further, since structural hazards only cause a stall if the instructions are precisely placed such that they need the respective resource at the same time, they are much more difficult to capture by a compiler. It is then necessary for an instruction scheduler to maintain a list of critical instruction pairs together with the forbidden issue cycle differences (see e.g. [AGG98]). Hence, except for VLIW processors, structural hazards are usually rather resolved by hardware-based detection using additional logic called *pipeline interlocks* [HP07].

A third type of hazards are *control hazards* that are incurred by branching instructions. Since the actual execution path that the program will take at runtime is not known at the time of compilation, the result of a branching instruction is usually predicted. The corresponding instructions of the predicted execution path are then issued and, if the prediction turns out to be wrong, a delayed issue of the correct instructions will take place. Different approaches exist to deal with the unknown runtime execution path from a compile-time perspective. One strategy is to first decompose a program fragment at branching instructions and all places that the control flow may jump to. The resulting branch-free subseries of instructions are called *basic blocks* and the task to schedule each basic block individually is referred to as *basic-block instruction scheduling* or *local instruction scheduling* [ALSU86]. This emphasizes a good performance for each independent program fragment independent from the execution path taken at runtime. However, especially for multiple-issue processors, it can be beneficial to allow the semantic-preserving motion of instruction across basic block boundaries since this may increase the possible amount of instruction-level parallelism. This general concept is referred to as *global instruction scheduling* [ALSU86] and may be refined in several ways. For example, one may try to optimize for the most probable execution path of a program, possibly by taking traces into account. Alternatively, in the context of real-time embedded systems, one might optimize for the execution time of the worst-case execution path [LM10].

Instruction scheduling is a central point for optimizations aiming at a sped up execution of programs. However, as already indicated, it is heavily interdependent with register allocation. Before register allocation, variables and temporary values used in the program are assigned to virtual registers that can exceed the number of available physical registers. While instruction scheduling deals with the assignment of execution units and issue cycles to instructions, register allocation deals with the assignment of physical registers to virtual ones. If the number of physical registers does not suffice in order to map all the used virtual registers to them without invalidating results, so-called *spill code* needs to be inserted into the program. These are additional stores and loads that temporarily move register contents to and from memory. Scheduling only prior to register allocation cannot take these additional instructions into account and could also lead to invalid results if the processor lacks pipeline interlocks. Scheduling only after register allocation would lead to avoidable dependencies being added by the register allocator. Hence, instruction scheduling is usually done twice, once before and once after register allocation [ALSU86], making this task even more important.

2.2 The Local Instruction Scheduling Problem

In the following, we strive to optimally solve the local instruction scheduling problem for a pipelined single-issue processor. In our context, an optimal solution minimizes the *makespan* that is defined as the completion time of the last instruction [BEP$^+$07].

2.2.1 Preparations

The assumed properties of a processor have several immediate implications. Since the processor is pipelined, the processing time of each instruction (apart from NOPs that we may count separately) is uniform. Further, we assume that exactly one new instruction can be issued every clock cycle. This allows to normalize all processing times to one since we need not consider the processor to be occupied for different durations depending on the concrete instruction. In the literature (see, e.g., [GLLK79]), this is usually referred to as *unit processing times*. As an immediate consequence and with a similar abstraction of NOPs, the completion time of an instruction is simply a constant number of cycles later than its starting time.

In our context, a *schedule* is a mapping of each instruction to a unique clock cycle where it is issued. For ease of language, we will also call this clock cycle the *issue cycle* of the instruction or simply say that an instruction is *scheduled* at this point in time. A schedule starts at cycle zero by inserting the first instruction into the pipeline and ends by inserting the last instruction into it. We consider the makespan to be the issue time of the last instruction plus one cycle (its normalized processing time). However, since this does not alter the optimization goal under the assumptions made, we will sometimes also use the term makespan when referring to the starting time of the last instruction. As already mentioned, there is no difference w.r.t. the makespan whether a NOP is inserted before issuing an instruction that would otherwise cause a stall or to stall the pipeline for one cycle after issuing the instruction. The number of NOPs exactly corresponds to the number of cycles additionally needed to complete the execution of a given set of instructions. Hence, the minimization of the makespan is equivalent to the minimization of NOPs.

It is of course possible to consider the problem at hand much more generally than in the context of instruction scheduling since it captures all settings that ask for an optimized permutation and starting time assignment for a set of given tasks with dependencies and minimum or fixed delays between each other.

2.2.2 Formal Problem Statement

In the local instruction scheduling problem, we are given a basic block that resembles precedence-constrained instructions and the associated latencies. As already discussed, a basic block does not comprise any side entrances or side exits. Further, the underlying precedence relationship must be acyclic in order to allow for a feasible sequencing of the instructions. More formally, we define the ISP as follows:

Definition 2.2.1. *(Local Instruction Scheduling Problem) Given a set of uni-cycle instructions I and an acyclic precedence relationship $R \subset I \times I$ along with a latency function $\ell : R \to \mathbb{N}_0$, compute a schedule $\sigma : I \to \mathbb{N}_0$ of the instructions I respecting all the precedence relations R and latencies ℓ and whose makespan $M = 1 + \max\{\sigma(i) \mid i \in I\}$ is minimum.*

In the literature, the latencies are often also called *delays* and the problem may also be named *single-machine scheduling under delayed precedence constraints*. Using the widely accepted notation proposed in [GLLK79], this problem can be classified as $1|\text{prec}(l_{ij}), p_j = 1|C_{max}$ or $1|\text{prec}(\text{delays}), p_j = 1|C_{max}$.

2.2.2.1 A Note on Latencies

The notion of latencies used in this thesis complies to the one used in some older articles about instruction scheduling (e.g. [BG89, PS93]) but differs from those used in more recent articles targeting also multiple-issue processors, in particular [WLH00, vBW01, MMvB08].

For a precedence $(i, j) \in R$ with associated latency $\ell(i, j)$, if t_i is the cycle where instruction i is scheduled, then j can be scheduled earliest in cycle $t_i + \ell(i, j) + 1$ (and not $t_i + \ell(i, j)$). In other words, the latency is the minimum number of cycles that indeed need to be in between the issue cycles of the two instructions and *not* the minimum issue cycle difference.

The definition used in the multiple-issue context stems from the fact that two instructions with a write after read dependency can typically be issued in the same clock cycle since the read will take place before the write in the pipeline. However, for single-issue processors, it leads to the peculiarity that there is no semantic difference between a zero- and a one-latency which is why the older definition is preferred. This decision also simplifies some of the discussions concerning instructions located between two dependent instructions that will be carried out in Chapter 3.

2.2.3 Complexity

If the latencies occurring are allowed to be arbitrary nonnegative integers, then the problem to find a schedule minimizing the makespan is strongly $\mathcal{NP}$-hard [HG83, FL96]. However, if the latencies are restricted to be zero or one, then the problem with n instructions is optimally solvable in time $\mathcal{O}(n^2)$ by using the algorithm of Bernstein and Gertner [BG89]. Their solution is based on a modification of the algorithm of Coffman and Graham [CG72] for optimally scheduling precedence-constrained instructions on two processors (without latencies and with unit processing times), $P2|\text{prec}, p_j = 1|C_{max}$. As is indicated by this result, there exists an immediate relationship between *scheduling precedence-constrained tasks on a single processor with latencies bounded by at most L cycles* and *scheduling precedence-constrained tasks on $L+1$ processors (without latencies)* that has also been observed by Hennessy and Gross [HG83]. As far as this is known to the author, it is still open at the time writing this thesis whether the problem is $\mathcal{NP}$-hard for fixed L, $L > 1$.

2.2.4 Results and Solution Approaches

We restrict our attention to research dealing with optimal basic-block instruction scheduling since the general literature on scheduling is inexhaustible.

An early integer programming approach targeting vector processors with multiple functional units but only a single instruction to be issued per clock cycle was proposed by Arya [Ary85]. The experiments were however restricted to three instances with up to 36 instructions and the optimization process needed to be interrupted for the larger two of them. One of the first constraint programming approaches to instruction scheduling was given by Ertl and Krall [EK91]. Their algorithm has been developed for a particular processor with latencies in the range $[1, 6]$ (Motorola 88100) and is shown to optimally schedule basic blocks with up to 20 instructions in reasonable time. However, the results indicate that the running times do not scale well for larger instances. Vegdahl [Veg92] proposed a dynamic programming algorithm for instruction scheduling. In this model, however, the emphasis is on simple precedences of instructions within a loop and latencies must be artificially represented by inserting dummy instructions - meaning that no other real instructions could be scheduled in between. This is therefore a different problem as considered here. Although the yet mentioned early exact approaches could only be applied to small problem instances, several attempts were made to integrate instruction scheduling with register allocation which considerably enlarges the solution space. Not surprisingly, the proposed models could also be applied to small instances only, e.g., the integer programming approaches proposed by Wilson et al. [WMGB93], Chang et al. [CCK97], and the enumerative methods by Chou and Chung [CC95], and Haga and Barua [HB01]. A good overview over further integrated approaches developed prior to 2000 by Gebotys and Elmasry [GE91] as well as by Zhang [Zha96] and strategies to use them in order to obtain good approximate solutions can be found in Kästner's thesis [Käs00]. Leupers and Marwedel [LM97] suggest an integer programming formulation to schedule a set of register-transfer-level operations on a restrictive parallel architecture as it can be found, e.g., in digital signal processors. As a seldom found feature, they incorporated side effects on live register values and limitations on parallel execution due to resource conflicts stemming from incompatible instruction formats into their model. However, it imposes a large number of variables and the authors report running times in the order of minutes already for comparably small sized problem instances. A similar integration of restrictions present in real-world DSPs concerning code selection, instruction scheduling, and register allocation was also studied by Bashford and Leupers targeting near-optimal solutions using constraint programming techniques [BL99]. Bednarski and Kessler [KB06] present a larger set of experiments incorporating both dynamic programming and integer programming approaches in order to integrate scheduling and register allocation. Again, high solution times or memory exceedances already for instances with 20 to 50 instructions are reported for both of the methods. A more recent survey on combinatorial approaches to standalone and integrated approaches to (local and global) instruction scheduling and register allocation is given by Castañeda and Schulte [LS14]. Despite the limited applicability of integrated approaches in the past, new sophisticated models and effective search space reduction techniques have recently led to a novel constraint programming approach [LCBS14] delivering near-optimal solutions.

The most recent contribution to attack the instruction scheduling problem with integer programming was given by Wilken, Liu and Heffernan [WLH00] in 2000. They were the first to optimally schedule a larger set of basic blocks with up to

$1,000$ instructions by applying some search space reduction techniques on the one hand, and cutting planes on the other. However, their experiments were restricted to instances with latencies in the range between zero and two clock cycles which does not conform to most real-world architectures. Also, their method is not well scalable since the number of variables and constraints depends on an upper bound on the makespan, and is therefore pseudo-polynomial. Later it was then shown that the method is not as successful on instances with larger and varying latencies [MMvB06]. An important result of their work is however that the reduction techniques are essential to be able to schedule real-world instances to optimality. One year later, van Beek and Wilken [vBW01] proposed a constraint programming approach that could optimally schedule the instances used for the experiments by Wilken, Liu and Heffernan even faster. After Heffernan and Wilken then proposed a set of methods to even more effectively reduce the search space of basic block instances [HW05] in 2005, Malik, McInnes, and van Beek [MMvB06, MMvB08] were able to improve their CP approach to solve the problem also for multiple-issue processors on an even larger set of instances (about $350,000$ basic blocks with up to $2,600$ instructions). While the previous solvers from [WLH00] and [vBW01] could not solve hundreds of these instances to optimality, there is only one instance that could not be solved by the CP solver within a time limit of ten minutes of CPU and system time for single-issue processors in our experiments presented in Sect. 3.7.

2.2.5 Motivation for a New Integer Programming Approach

The successful application of the constraint programming solver from [MMvB06, MMvB08] to a large set of instances and more complex multiple-issue processors has been the largest advance on solving the ISP so far. Taking a closer look on their approach, however, makes apparent that many of the search space reductions that are key to the success of their solver are not specific to constraint programming and some of them may also be improved as we will discuss in detail in Sect. 2.4. As just discussed, the previously best-performing IP solver published prior to the most recent preprocessing advances was not capable to handle a large benchmark set comprising instances with varying and large latencies. In general, it is interesting from a research point of view whether integer programming methods can be equally successful when they are combined with the search space reduction techniques used before. Further, the existing IP models were all either based on time-indexed formulations with binary variables or on disjunctive formulations with general integer variables as is addressed in Sect. 3.1. As far as this is known to the author, there has yet been no practically implemented method that tries to tackle the problem by interpreting the ISP as a particular *ordering problem*. This was the final motivation to design a completely novel integer programming approach that can deal with arbitrary latencies and takes the mentioned preprocessing techniques into account. This approach, that is presented in Chapter 3, leads to a new characterization of the feasible solutions to the problem. It allows for the application of polyhedral knowledge from the mathematical optimization community which is also seldom observed in the context of this problem (although we do not want to overlook that Wilken, Liu, and Heffernan separate fractional LP solutions using problem-specific cutting planes).

Finally, it is well conceivable that the new approach may be applicable to related problems with different objective functions or at least inspire similar formulations for these problems. The new approach will be subject to Chapter 3. For now, we will turn the attention to the most important definitions and notations, and to some central results and search space reduction techniques.

2.3 Basic Concepts and Notations

2.3.1 Predecessors, Successors, Independence

For each precedence relation $(i, j) \in R$, we call i a *predecessor* of j and j a *successor* of i. Precedence relations are transitive, i.e., if $(i, j) \in R$ and $(j, k) \in R$, then i is also a predecessor of k and k is a successor of i. We will denote the transitive closure of R with R^*. For ease of notation, if i (j) is a predecessor (successor) of j (i), we also denote this by $i \prec j$ $(j \succ i)$. If neither $i \prec j$ nor $j \prec i$, then i and j are said to be independent (from each other) and we will sometimes denote this by $i \parallel j$. For ease of reference, we will denote the *immediate* predecessor (successor) set of an instruction v by $P(v)$ $(S(v))$. If we want to refer to the entire (transitive) predecessor (successor) set of an instruction, we will write $P^*(v)$ $(S^*(v))$ instead.

2.3.2 Data-Dependency Graphs

A basic block is usually modeled as a data-dependency DAG $G = (V, A)$ along with a weight function $w : A \to \mathbb{N}_0$ where the vertices V identify the instructions I and there is an arc $(i, j) \in A$ for each $(i, j) \in R$ with weight $w(i, j) = \ell(i, j)$. Each vertex with no predecessor in G is said to be a *source* of G. Similarly, each vertex with no successor in G is called a *sink* of G. We will assume that a DAG is normalized to have a single *super source* $b \in V$ and *super sink* $e \in V$. A super source (sink) has a leaving (entering) arc to each source (from each sink) with zero weight. Fig. 2.3 shows an example.

Throughout this thesis, we will treat instructions and their corresponding vertices, precedences and their corresponding arcs as well as latencies and their corresponding arc weights interchangeably, thereby interpreting the super source and super sink as pseudo-instructions that can be removed from the computed schedule afterwards without altering optimality. We will also refer to data-dependency DAGs as dependency DAGs or even just as DAGs. Further, analogously to the precedence relationships R, we will denote the transitive closure of the arc set A with A^*.

2.3.2.1 Critical Paths and Distances, Transitivity

Let $G = (V, A)$ be a dependency DAG and $i, k \in V$ such that $i \prec k$. Consider a simple path $P = i \to j_1 \to \cdots \to j_p \to k$ from i to k with $p \geq 0$ intermediate vertices in G. We refer to P's vertices by $V(P)$ and to its arcs by $A(P)$. The length of P is given by the sum of its arc weights plus the number of intermediate vertices, i.e.,

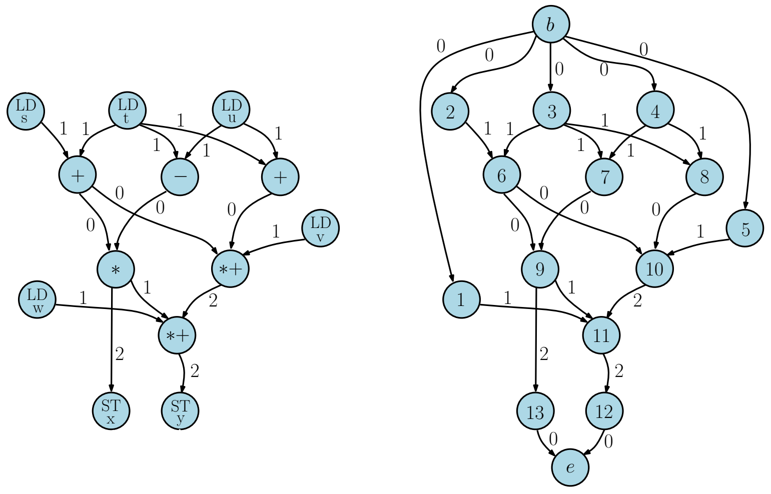


Figure 2.3: An example data-dependency DAG for a basic block consisting of the computations $x = (s+t)*(u-t)$ and $y = w+(x*(v+((s+t)\cdot(t+u)))$ assuming the presence of multiply-accumulate instructions (left) and its normalized version with a topological numbering of its vertices (right).

by $|V(P)| - 2 + \sum_{(i,j)\in A(P)} w(i,j)$. Any such path P induces a lower bound on the gap between i and k in any feasible schedule. Hence, a longest among such paths (called a *critical path*) between i and k in G imposes the tightest such lower bound, the *critical path distance* $cp(i,k)$.

There are several ways to obtain good lower bounds on the minimum number of cycles between two instructions apart from considering paths only. Some of them will be discussed in Sect. 2.4. To be unambiguous in notation w.r.t. the critical path distances implied by the given dependency DAG $G = (V, A)$, we introduce a more general and intuitive concept of *distances*, and associate such a distance $d_{i,k}$ with every pair of instructions $i, k \in I, i \neq k$. At each point in time, the distance $d_{i,k}$ reflects the best known lower bound on the gap between i and k, regardless how it has been obtained. We make the convention that $d_{i,k}$ is greater than or equal to zero if $i \prec k$ and set to $-\infty$ otherwise. Clearly, it holds that $cp(i,k) \geq \ell(i,k)$ for each $(i,k) \in A$ and $d_{i,k} \geq cp(i,k)$ for all $(i,k) \in A^*$. Naturally, if some particular distance $d_{i,k}$ can be improved, this can be equally exploited to transitively propagate distance information as when considering critical paths. In particular, for any path P as described above, $d_{i,k} \geq |V(P)| - 2 + d(P)$ where $d(P)$ is the sum of the distances along the path. More generally, each valid set of distance lower bounds for a given instance can itself be interpreted as a dependency DAG that must have the same optimum makespan as the original DAG. Further, the distance $d_{b,e} \geq cp(b,e) = cp(G)$ can be used to derive a lower bound on the makespan.

If $i \prec j$, $j \prec k$, and $d_{i,k} = d_{i,j} + 1 + d_{j,k}$, the distance lower bound between i and k is redundant information. In case that $d_{i,k} > d_{i,j} + 1 + d_{j,k}$ holds for all $j \in I$, then the distance relation between i and k is not redundant (though the precedence relation is). Also, the distances $d_{i,j}$ and $d_{j,k}$ may remain nonredundant since they may still partially characterize the position that j has to attain in between i and k.

2.3.2.2 Regions

The following notion of sub-DAGs called *regions* [WLH00] is useful.

Definition 2.3.1. (Region [WLH00]). *Let $G = (V, A)$ be a dependency DAG and $s, t \in V$ such that there are at least two vertex-disjoint paths between s and t. Define $V_{s,t}$ to be the set of vertices reachable on any s-t-path in G and $A_{s,t}$ as the union of the arcs of all these paths. Then the DAG $G_{s,t} = (V_{s,t}, A_{s,t})$ is called a region of G.*

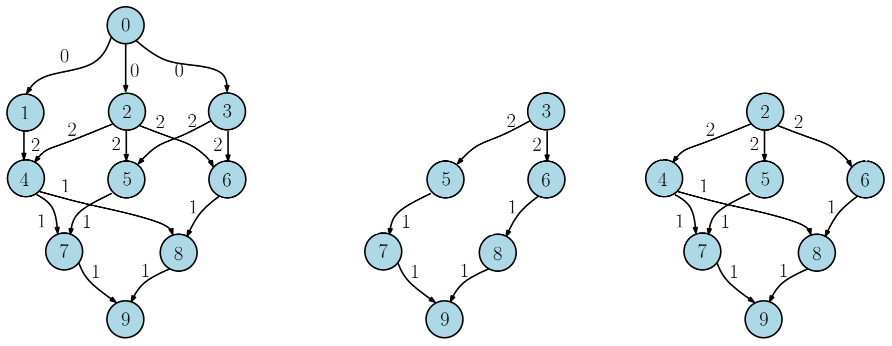


Figure 2.4: A DAG and two example regions $G_{3,9}$ and $G_{2,9}$.

Regions can be nested. For example, the small region $G_{2,7}$ (which is not shown explicitly) is part of the region $G_{2,9}$ in Fig. 2.4. In addition, regions may be classified as single-entry-single-exit (SESE) regions and non-SESE regions. A region $G_{s,t} = (V_{s,t}, A_{s,t})$ is said to be a SESE region if there is no vertex $v \in V_{s,t} \setminus \{s, t\}$ with a predecessor $u \notin V_{s,t}$ (side-entry vertex) or successor $w \notin V_{s,t}$ (side-exit vertex). Wilken et al. proved the following claims connected to SESE and non-SESE regions.

Lemma 2.3.2. ([WLH00]). *Let $G_{s,t} = (V_{s,t}, A_{s,t})$ be a SESE-region of a DAG $G = (V, A)$. Let σ be a schedule of $V_{s,t}$ with associated order π. Then the order π can be fixed for schedules of V without altering optimality if the following conditions hold:*

- *σ is a dense schedule, i.e., it does not contain any NOPs.*
- *If $\pi(u) = \pi(s) + 1$, then $w(s, u) = \min\{w(s, v) \mid (s, v) \in A_{s,t}\}$.*
- *If $\pi(u) = \pi(t) - 1$, then $w(u, t) = \min\{w(v, t) \mid (v, t) \in A_{s,t}\}$.*

Lemma 2.3.3. ([WLH00]). *Let $G_{s,t} = (V_{s,t}, A_{s,t})$ be a non-SESE-region of a DAG $G = (V, A)$. Let σ be a schedule of $V_{s,t}$ with associated order π. Then the order π can be fixed for schedules of V without altering optimality if the following conditions hold:*

- *σ satisfies the conditions for SESE regions from Lemma 2.3.2.*
- *If v is a side-exit vertex, then all vertices u with $\pi(u) < \pi(v)$ must be predecessors of v.*
- *If v is a side-entry vertex, then all vertices w with $\pi(v) < \pi(w)$ must be successors of v.*

Lemmata 2.3.2 and 2.3.3 may be used to replace regions of a DAG by linear sequences of vertices by, e.g., heuristically scheduling them and testing whether the conditions for the respective type of region are satisfied.

2.3.3 Definitions Related to Lower and Upper Bounds

We denote the global lower and upper bounds on the optimum makespan M^* by M_{lb} and M_{ub}. Furthermore, we consider lower and upper bounds on the issue cycles of each instruction $i \in I$ and denote them by lb_i and ub_i. In the literature, the lower bounds lb_i are also referred to as $ASAP$ (as soon as possible) time steps. Similarly, the upper bounds ub_i are referred to as *reverse lower bounds*, or $ALAP$ (as late as possible) time steps. The interval $[lb_i, ub_i]$ will frequently be referred to as the *(scheduling) range* of an instruction $i \in I$.

We assume the super source $b \in V$ to be assigned clock cycle zero. While lower bounds on the issue cycles of instructions, and thus also an initial M_{lb}, can be directly determined from the best known distance of an instruction from b, upper bounds on the issue cycles are always related to a predetermined global upper bound M_{ub}. This becomes apparent in the following definitions and relations:

$$\begin{aligned}
lb_b &= ub_b = 0 && \\
lb_e &= M_{lb} - 1 && \\
ub_e &= M_{ub} - 1 && \\
lb_i &= d_{b,i} + 1 && \text{for all } i \in V \setminus \{b, e\} \\
ub_i &= M_{ub} - d_{i,e} - 1 && \text{for all } i \in V \setminus \{b, e\} \\
lb_k &\geq lb_i + d_{i,k} + 1 && \text{for all } (i,k) \in A^* \\
ub_i &\leq ub_k - d_{i,k} - 1 && \text{for all } (i,k) \in A^*
\end{aligned}$$

2.4 Search Space Reduction Techniques

An effective restriction of the search space is key to every exact algorithm, especially if the problem to be solved is known to be $\mathcal{NP}$-hard. In case of the ISP, we are interested in global lower and upper bounds on the schedule length on the one hand, and in particular lower and upper bounds on the schedule position of each instruction on the other. Once a global upper bound M_{ub} has been determined, valid initial upper bounds on the issue cycles of all instructions can be established using the relation $ub_i = M_{ub} - d_{i,e} - 1$ as described in Sect. 2.3.3. In addition to the discussion of existing search space reductions, we will point out improvements on some of these techniques. We will also derive new methods to improve on bounds and distances, and to find new precedence relationships without altering the optimum schedule length. These methods can, in principle, be applied with any instruction scheduling approach. Further techniques that are however inherently related to the novel integer programming models for the ISP developed in this thesis will be discussed separately in Chapter 3.

2.4.1 Global Upper Bounds

The length of any feasible schedule provides an upper bound on the makespan of an optimum schedule. The predominant heuristic method to derive feasible schedules is *list scheduling* [Gra69]. List scheduling subsumes a generic class of algorithms that select instructions from a list of ready ones. We call an instruction i *ready* if (a) all predecessors of i are already scheduled and (b) all latencies w.r.t. i are satisfied in the currently considered clock cycle. If only (a) holds, we will sometimes call the respective instructions *waiting* (to become ready).

List scheduling is flexible since it can easily handle parallel or multiple-issue processors and different priority functions to decide which instruction to schedule next when more than one is ready. It is a well-known result [Gra69] that, regardless how priorities are assigned, list scheduling for precedence- (but not latency-) constrained instructions with arbitrary processing times is a $\left(2-\frac{1}{m}\right)$-approximation algorithm for $m \geq 2$ processors. For the problem considered here ($m = 1$, unit processing times, precedences and latencies), Bernstein, Rodeh, and Gertner [BRG89] show that list schedules are no worse than $2 - \frac{1}{L+1}$ times the optimum where L is the maximum latency occurring. Palem and Simons [PS93] generalized their result to an approximation ratio of $2 - \frac{1}{m(L+1)}$ for m processors again. We study list scheduling more detailed for single processors in the subsequent section.

2.4.1.1 List Scheduling

The following is a close-to-implementation description of a general list scheduling algorithm for single processor scheduling. In this implementation, priority queues rather than lists are used. It assumes the existence of a priority queue operation `return_min()` that, in contrast to the usual operation `extract_min()`, returns the current minimum object but does not remove it from the queue. The implementation makes use of two priority queues, one that holds the set of ready instructions and another one that holds the set of waiting instructions.

The procedure, that is depicted as Algorithm 1, starts by assigning the first cycle to the artificial super source instruction. After issuing any instruction i, the earliest starting times of successor instructions s of i are updated based on the associated latencies. If any successor s is discovered whose predecessors are all already scheduled, the instruction is temporarily stored in the waiting queue to be efficiently retrieved when it becomes ready. Furthermore, this allows to 'jump' to the earliest clock cycle in which an instruction will become ready if there is currently no ready instruction. The keys of the instructions in the ready queue are their priorities and the keys in the waiting queue are the earliest starting times. At each considered clock cycle c instructions that become ready at c are inserted into the ready queue. Then, a highest-priority instruction is selected from the ready queue and issued. The described process is iterated until all the instructions are scheduled. NOPs are not scheduled explicitly, since otherwise the running time would depend on the schedule length and not only on the size of the input data.

Algorithm 1 List Scheduling on a Single Processor

```
function ListSchedule(DAG $G = (V, A)$ with latencies $\ell$, Array of priorities $P$)
    $EST$                                   # Array of earliest starting times for each $v \in V$
    $CYC$                                   # Array of issue cycles for each $v \in V$
    $R \leftarrow \emptyset$                # Priority queue of ready instructions
    $Q \leftarrow \emptyset$                # Priority queue of waiting instructions
    $c \leftarrow 0$
    $R.insert(v_0, 0)$                      # Enqueue the super source
    while $R \neq \emptyset$ or $Q \neq \emptyset$ do
        while $Q \neq \emptyset$ and $(Q.return\_min() == c)$ do
            $i \leftarrow Q.extract\_min()$
            $R.insert(i, P[i])$             # Insert instructions that become ready at $c$
        if $R \neq \emptyset$ then
            $i \leftarrow R.extract\_min()$
            $CYC[i] \leftarrow c$           # Schedule $i$ at cycle $c$
            for all successors $s$ of $i$ do
                if $c + \ell(i, s) + 1 > EST[s]$ then
                    $EST[s] \leftarrow c + \ell(i, s) + 1$   # Update earliest starting times
                if all predecessors of $s$ scheduled then
                    $Q.insert(s, EST[s])$   # Insert instruction into waiting queue
        if $R \neq \emptyset$ then          # Still ready instructions.
            $c \leftarrow c + 1$            # Just increment $c$.
        else if $Q \neq \emptyset$ then     # No ready, but waiting instructions.
            $c \leftarrow EST[Q.return\_min()]$   # Increase $c$ to minimum EST of waiting instr.
```

The amortized worst-case running time of this algorithm is $\mathcal{O}(|V| \log |V| + |A|)$ since each instruction is inserted and removed from a priority queue at most twice and each arc of the DAG is scanned exactly once in order to update the earliest starting times of successors and potentially release them.

There exist several strategies to determine priorities. The most common and also a very successful strategy is to assign each instruction a priority that is equal to its critical path distance to the sink. Hence, if multiple instructions are ready, one of those with the smallest upper bound on its issue cycle is selected. In the literature, this strategy is often referred to as *critical path list scheduling.* Many computational experiments (see, e.g., [EK91, MMvB08, Mal08, WLH00]) reveal near-optimal performance of critical path list scheduling when averaging results over a particular set of instances. The fine-grained results in [MMvB08, Mal08] show, however, that the number of basic blocks where list scheduling does not find optimal schedules grows significantly with increasing size of the instances (up to 20% for basic blocks with more than 250 instructions in their results). Another idea is to perform list scheduling in a *backward* fashion which basically amounts to a reversion of all the arcs [CSS98]. This way, instructions are potentially rather clustered around the sink instead of around the source. There are instances where this leads to better solutions, but the backward method can of course also perform worse than the standard one.

2.4.1.2 Excursion: A New Result On Coffman-Graham List Schedules

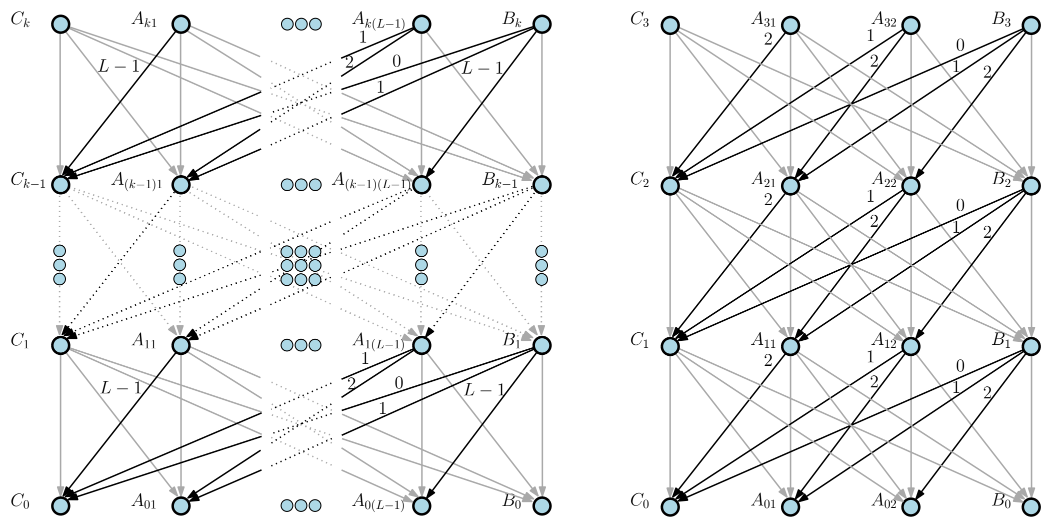
There is a subclass of list scheduling algorithms that partition the set of instructions into *levels* and then assign priority *labels* to the instructions. One instance of this class is the algorithm of Coffman and Graham [CG72] that optimally schedules precedence-constrained instructions on two processors (without latencies and with unit processing times). A modified version by Bernstein, Rodeh, and Gertner [BRG89] that takes latencies into account (and that is here reinterpreted with latencies associated to arcs instead of instructions) is as follows.

1. Set the level $h(t)$ of the super sink to zero. For each other instruction $i \in I \backslash \{t\}$, recursively compute its level as $h(i) = \max\{h(s) + \ell(i,s) \mid s \in S(i)\}$. This partitions I into m subsets I^a, $0 \leq a \leq m-1$ such that for all $i \in I^a$, $h(i) = a$.
2. Process the subsets I^a in increasing order of a. Suppose the labels $1, \dots, k-1$ have already been assigned to the instructions in I^b, $0 \leq b < a$. Then, assign the labels $k, \dots, k + |I^a| - 1$ to the instructions in I^a as follows: For each $i \in I^a$ whose successors $s \in S(i)$ have all been already labeled, let $\Omega_i = \{\lambda(s) \mid s \in S(i)\}$. Sort the values in each Ω_i in decreasing order. Find an instruction $x \in I^a$ such that Ω_x is lexicographically minimum. Set $\lambda(x) = k$.
3. Create a priority list placing instructions with a higher label first.
4. Perform a list schedule using the priority list.

For the special case that all the latencies are either L or zero, Bernstein, Rodeh and Gertner show that this algorithm has a worst-case performance of $2 - \frac{2}{L+1}$ times the optimum [BRG89]. This is slightly better than the worst-case performance $2 - \frac{1}{L+1}$ of general list schedules. In particular, the algorithm is optimal for $L = 1$. Without reference to the cited article, the optimality result for $L = 1$ was extended to instructions with arbitrary processing times by Finta and Liu [FL96]. However, for the unit processing times case, it has been left open whether the better approximation ratio still holds if latencies in $\{0, \dots, L\}$ (instead of only $\{0, L\}$) are allowed. We may answer this question in the negative sense by providing a corresponding worst-case instance.

Theorem 2.4.1. *If arbitrary latencies from the range $\{0, \dots, L\}$ are permitted, then the above listed algorithm by Bernstein, Rodeh and Gertner has a tight worst-case approximation ratio of $2 - \frac{2}{L+1}$.*

Proof. For a maximum latency of L, the instance schematically depicted (without super source and sink to ease the description) in Fig. 2.5 has exactly $n = (L+1)(k+1)$ instructions. Here, $k > 0$ is a scaling parameter that controls the number of different levels that the above algorithm will assign to the instructions. The instance is constructed such that vertices in the same row will be assigned the same levels by the algorithm. Look at the precedence structure in Fig. 2.5. There are exactly $k+1$ rows each of which has exactly $L+1$ vertices. To not overload the image, not all



(a) Worst-case instance construction scheme. **(b)** A concrete example for $L = 3$

Figure 2.5: Worst-case instances for Coffman-Graham list schedules.

latencies associated to the precedence arcs are printed. Every weakly solid drawn arc (downward or rightward) has latency L and every dark (leftward) arc has the latency associated that is drawn next to the left of it.

In general, for $j \geq 1$, instruction B_j has an arc with latency L to vertex B_{j-1} and latency zero to C_{j-1}. Further, it has arcs with latency x to each A_{j-1x}, $1 \leq x \leq L-1$. For any two vertices A_{jx} and A_{j-1y} with $y < x$ there is an arc from A_{jx} to A_{j-1y} with latency $L-(x-y)$. The same applies to arcs (A_{jx}, C_{j-1}), i.e, the arcs have the associated latency $L-x$. Each vertex in row $j \geq 1$ has an L-weighted arc to every vertex in row $j-1$ with a right-or-equal position. For this reason, the levels $h(i)$ derived by the above algorithm will be jL for all vertices in row $0 \leq j \leq L$.

To construct a worst-case schedule, we start with the instructions i on level (in row) zero. According to the second step of the algorithm, we may choose their $\lambda(i)$ arbitrarily. We set $\lambda(C_0) = 1$, $\lambda(A_{01}) = 2$, …, $\lambda(B_0) = L+1$. Now consider the Ω-sets for the vertices on level L (in row one). The vertices B_1 and $A_{1(k-1)}, \ldots, A_{11}$ and C_1 are all connected to all vertices on level zero, hence $\Omega(i) = \{L+1, \ldots, 1\}$ for all of them. Again, as all of these sets are lexicographically equal, we may choose the λ-labels on level one arbitrarily and set them in the same manner as for level zero. By applying this scheme until the last level, we obtain a complete labeling.

By inspecting the instance, one can see that, if the instructions are processed in the inverse order of their λ-values, then an optimal schedule without any NOPs results. However, if the vertices are scheduled in λ-order, then after each level there will be exactly L NOPs since C_j (that is always scheduled last in its row) imposes a latency of L on all vertices of level $j-1$.

In order to prove that this leads to a worst case performance of $2 - \frac{1}{L+1}$ times the optimum, we apply the same arguments as in [BRG89]. Clearly, the optimum makespan C_{opt} is $n = (k+1)(L+1)$ while the constructed schedule will introduce kL

NOPs, i.e., $C_{cg} = n + kL$. We evaluate the ratio R between both schedule lengths as follows.

$$R = \frac{C_{cg}}{C_{opt}} = \frac{kL + n}{n} = \frac{kL + (k+1)(L+1)}{(k+1)(L+1)} = 2 - \frac{k + L + 1}{(k+1)(L+1)}$$

The last equation holds since $(k+1)(L+1) + kL + k + L + 1 = (k+1)(L+1) + k(L+1) + L + 1 = (k+1)(L+1) + (k+1)(L+1) = 2(k+1)(L+1)$. Hence, by increasing k (the number of levels), R can be made arbitrarily close to $2 - \frac{1}{L+1}$. $\square$

Moreover, the instance shows that the ratio still holds when using the numbers of successors or a combination of these with the critical path lengths as the priority labels.

2.4.2 Global Lower Bounds

2.4.2.1 Trivial Critical Path Lower Bound

As already indicated, a lower bound on the distance between the super source and the super sink of a dependency DAG G implies a lower bound on the makespan, namely $M_{lb} \geq d_{b,e} + 2 \geq cp(b, e) + 2$. This lower bound can sometimes be improved by a simple observation. Let $P = b \to i_1 \to \cdots \to i_{|V(P)|-2} \to e$ be a source-sink path in G. As defined in Sect. 2.3.2.1, its length is given by the sum of the distances $d(P)$ along the path plus the number of immediate vertices $|V(P)| - 2$ between the source and the sink. Consequently, the number of vertices not contained in P is $k = |V| - |V(P)|$ and these vertices may ideally be used to cover some of the latencies $d(P)$. However, if $k > d(P)$, then some of the k vertices will contribute additionally to the schedule length. As a simple formula, this yields:

$$M_{lb} \geq d_{b,e} + 2 + \max\{0, k - d(P)\}$$

2.4.2.2 A Relaxation Technique by Rim and Jain

In 1994, Rim and Jain [RJ94] presented a relaxation technique to obtain lower bounds on the makespan of a given dependency DAG. We discuss the basic ideas while matching the presentation with our notation of lower and upper bounds on issue cycles. Their method exploits the already addressed fact that the upper bounds on issue cycles depend on M_{ub}. More generally, for any schedule of length M, instruction i must start at time $ub_i^M = M - d_{i,e} - 1$ the latest. Hence, setting $M = M_{lb}$ makes it possible to obtain upper bounds $ub_i^{M_{lb}}$ on issue cycles that must be respected if the schedule length M_{lb} shall be realized. Conversely, if we solve the (relaxed) problem and find an instruction i that is assigned a cycle $c > ub_i^{M_{lb}}$, then M_{lb} can be improved by the respective amount of violation.

We restate the associated mathematical problem formulation of the relaxed scheduling problem by introducing variables $x_{i,t}$ that attain value one if instruction i is assigned to clock cycle $t \in T$ and zero otherwise. For our considerations, it is sufficient to have some (trivial) upper bound on the largest necessary clock cycle provided by T, e.g., the corresponding algorithm to solve this model cannot assign cycles to instructions that are larger than $M_{lb} + |I|$. We denote with $g = M^* - M_{lb}$ the gap between M_{lb} and the unknown optimum schedule length M^*. Since M_{lb} is a constant value, we can optimize for g instead of M^*. By the observations made above, it holds that $g \geq (\sum_{t \in T} t\ x_{i,t}) - ub_i^{M_{lb}}$ for each instruction $i \in I$. Rearranging this inequality to $\sum_{t \in T} t\ x_{i,t} \leq ub_i^{M_{lb}} + g$, and adding the according inequalities for lower bounds as well, one obtains the following relaxed problem formulation:

$$
\begin{array}{lll}
\min & g & \\
\text{s.t.} & \sum_{t \in T} x_{i,t} = 1 & \text{for all } i \in I \\
& \sum_{i \in I} x_{i,t} \leq 1 & \text{for all } t \in T \\
& \sum_{t \in T} t\ x_{i,t} \geq lb_i & \text{for all } i \in I \\
& \sum_{t \in T} t\ x_{i,t} \leq ub_i^{M_{lb}} + g & \text{for all } i \in I \\
& x_{i,t} \in \{0,1\} & \text{for all } i \in I,\ \text{for all } t \in T \\
& g \in \mathbb{N}_0 &
\end{array}
$$

In this relaxed scheduling problem, only lower and upper bounds, and the resource constraints (at most one instruction at a time) are respected, but not the latency constraints. Fortunately, this problem can be solved by a simple greedy algorithm (listed as Algorithm 2) that was also given by Rim and Jain [RJ94].

Algorithm 2 Greedy Algorithm by Rim and Jain

```
function RimJain($G = (V, A)$, $lb$, $ub^{M_{lb}}$)
    BucketSort($V, ub^{M_{lb}}$)          # Sort instructions in increasing order of $ub^{M_{lb}}$
    $g \leftarrow 0$
    for all $v \in V$ do
        Assign $v$ to the earliest free cycle $c \geq lb_v$
        if $c - ub_v^{M_{lb}} > g$ then
            $g \leftarrow c - ub_v^{M_{lb}}$
    return $g$
```

For ease of reference, we will refer to the schedules produced by this procedure as *Rim-Jain schedules.* Rim and Jain propose to compute lower bounds lb_i and M_{lb} based on the critical path information given from the DAG and to derive the corresponding upper bounds $ub_i^{M_{lb}}$ in this way. Hence, originally, the algorithm is applied with M_{lb} set to $cp(G)+2$. The authors state that the algorithm runs in time $\mathcal{O}(n + cp(G) \cdot C)$ in this case where C is the completion time of the schedule [RJ94]. However, the runtime should include the cardinality of the arc set of G since each arc needs to be scanned at least once in order to compute the critical path distances. A correctness proof of the algorithm is given in the original article. However, the correctness can also be verified by a simple argument. Let s_i be the clock cycle assigned to an instruction i. The optimal value of g is given by the largest difference $s_i - ub_i^{M_{lb}}$ among all i. Let x be an instruction assigned to s_x such that $s_x - ub_x^{M_{lb}} = g$. Suppose now that g is not optimal. Then it would be possible to realize a gap smaller than g for each instruction, in particular for any instruction that now has a gap of g. Consider x again. Clearly, a position $s'_x > s_x$ would result in a larger gap and cannot be optimal. However, since the instructions are processed in increasing order of their upper bound values, all clock cycles in the (possibly empty) range $[lb_x, s_x[$ are assigned to instructions with a smaller-or-equal upper bound value. Hence, exchanging any of them with x would also result in a larger-or-equal gap g. This verifies optimality of g.

In 1996, Langevin and Cerny [LC96] proposed to use the method by Rim and Jain recursively in a topologically ordered fashion. Their idea is to apply the original method to each sub-DAG induced by interpreting each predecessor of i as the sink prior to running the algorithm on the (sub-)DAG with sink i. This way, improved lower bounds on the issue cycle of all predecessors of each instruction $i \in I$ will be already respected when computing a lower bound on i's issue cycle. As the authors report, this leads to better global results in many cases while the runtime observed in practice increases only moderately since many of the considered sub-DAGs are small.

Within the solution approaches developed in Chapter 3, the recursive method is applied to DAGs that reflect the already strengthened distance information that is obtained during a preprocessing phase.

2.4.2.3 A New Result on Rim-Jain Schedules

In the original articles [RJ94, LC96], the authors describe the relaxation in terms of a negligence of the precedence and latency constraints. However, if the lower and upper bounds on the issue cycles being input to the algorithm are consistent with the precedence relation, in fact only the latency constraints will be relaxed when applying the algorithm. The main reason for this property is that the algorithm of Rim and Jain assigns positions to instructions in the order of increasing upper bounds. To prove this formally, we first clarify what is meant with *consistency* of bounds w.r.t. precedence relations.

Definition 2.4.2. (Consistency of bounds [BEP+07]). *Let I be a set of instructions and $R \subset I \times I$ be the precedence relations on I. The lower and upper bounds lb_i and ub_i on the issue cycles for all instructions $i \in I$ are called* consistent *(with the precedence relations R) if $lb_i + 1 \leq lb_j$ and $ub_i \leq ub_j - 1$ for each $(i, j) \in R$.*

Definition 2.4.2 can be read equally for dependency DAGs $G = (V, A)$. An analogous definition of consistency could be stated for latencies or distances by simply replacing $lb_i + 1 \leq lb_j$ and $ub_i \leq ub_j - 1$ by respectively $lb_i + d_{i,j} + 1 \leq lb_j$ and $ub_i \leq ub_j - d_{i,j} - 1$. We may now state the main theorem.

Theorem 2.4.3. *Let $G = (V, A)$ be a dependency DAG and let σ be a schedule for G obtained by the greedy algorithm of Rim and Jain using lower and upper bounds consistent with the precedences given by A. Let $\sigma(v)$ be the position of each $v \in V$ in σ. Then for each $(i, j) \in A$, it holds that $\sigma(i) < \sigma(j)$.*

Proof. Let $i, j \in I$ be two arbitrary instructions such that $(i, j) \in A$. Since the lower and upper bounds are consistent with the precedences given by A, it follows that $ub_i \leq ub_j - 1$ and $lb_j \geq lb_i + 1$. Since $ub_i < ub_j$, the algorithm processes i before j. It starts to look for a free cycle c at position $lb_i < lb_j$ and moves forward until it finds one. Either a position strictly before lb_j can be found or the algorithm proceeds to find a position greater or equal to lb_j and will stop at the first free cycle. So when j is processed, either $c < lb_j$ or the algorithm iterates over the same dense block of instructions that now contains already i (more precisely, the range $[lb_j, lb_i]$). Hence, in both cases j will be placed after i in σ. □

A different way to prove this result is by using the notion of *normal* schedules. We first restate the definition of normal schedules from [BEP$^+$07] in terms of our notation and then proceed with a lemma from the same reference.

Definition 2.4.4. (Normal schedules [BEP$^+$07]). *Let σ be a schedule of the instructions I and $\sigma(i)$ be the position of $i \in I$ in σ. If $\sigma(i) < \sigma(j)$ implies that either $ub_i \leq ub_j$ or $lb_j > \sigma(i)$ (or both), then σ is called a* normal *schedule.*

Lemma 2.4.5. ([BEP$^+$07]). *If the lower and upper bounds on issue cycles are consistent with the precedence relation, then any normal single-processor schedule that satisfies the lower and upper bounds must obey the precedence relation.*

We may now prove Theorem 2.4.3 via Lemma 2.4.5 by showing that Rim-Jain schedules are normal.

Theorem 2.4.6. *Let $G = (V, A)$ be a dependency DAG and σ a schedule for G obtained by the greedy algorithm of Rim and Jain using lower and upper bounds consistent with the precedences given by A. Then σ is normal, i.e, for each pair $i, j \in I$, $\sigma(i) < \sigma(j)$ implies that either $ub_i \leq ub_j$ or $lb_j > \sigma(i)$.*

Proof. Consider two arbitrary instructions $i, j \in I$ such that $\sigma(i) < \sigma(j)$. If $ub_i \leq ub_j$ or $lb_j > \sigma(i)$, there is nothing to show. So assume that both conditions do not hold. Hence, $ub_i > ub_j$ and j is processed by the algorithm prior to i. Since also $lb_j \leq \sigma(i)$, j is scheduled in cycle $\sigma(i)$ the latest since i is not processed yet and hence the cycle must be free at this point. This contradicts the assumption $\sigma(i) < \sigma(j)$ and therefore either $ub_i \leq ub_j$ or $lb_j > \sigma(i)$ or both must hold. □

This result is interesting in the following sense. The ISP mainly consists of four types of constraints: A solution must obey the alldifferent property, precedences, distances and integrality. Relaxing integrality basically amounts to solving the problem as a linear program. The algorithm of Rim and Jain maintains the alldifferent property and integrality but relaxes distance constraints. Since one cannot relax precedence constraints without relaxing distance constraints, it is basically the only other way to relax the problem at all if clock cycles shall not be assigned multiple instructions. And it is good news that it can be solved combinatorially by a simple greedy algorithm. One could come to the idea to try to make such a Rim-Jain schedule feasible for the original problem. Since Rim-Jain schedules do not violate precedences, this exactly corresponds to the idea to use the (inverse) order of the instructions as a priority for the list scheduling algorithm. Then at any time where multiple instructions are ready, the one that is positioned earlier in the Rim-Jain schedule will be selected, leading to a feasible schedule with the same order of instructions. While this is a straightforward idea, it has been observed experimentally that this strategy frequently leads to solutions that are far from being optimal.

2.4.2.4 A New Way to Further Exploit Rim-Jain Schedules

Suppose the lower bounding algorithm by Rim and Jain is run with global lower bound M_{lb} so that no instruction misses its corresponding deadline $ub_i^{M_{lb}}$, but that there is some instruction i with $lb_i \neq ub_i^{M_{lb}}$ placed exactly at its upper bound. By the construction of the algorithm, instruction i could not be placed earlier, i.e., in the interval $[lb_i, ub_i^{M_{lb}} - 1]$, due to a dense block of instructions that all have a smaller-or-equal upper bound.

In fact, the dense block of instructions could even start earlier, say at position x. So let there be such a dense range $[x, ub_i^{M_{lb}}]$. In case that there is an instruction j with $ub_j^{M_{lb}} > ub_i^{M_{lb}}$ but $lb_j \in [x, ub_i^{M_{lb}}]$, like illustrated in Fig. 2.6, then a reduction of j's upper bound below $ub_i^{M_{lb}}$ would immediately result in a later position of i and, therefore, lead to a new larger lower bound.

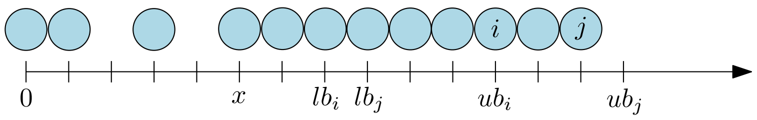


Figure 2.6: A situation in Rim-Jain schedules that can be exploited to improve lower bounds.

This immediately leads to the following theorem.

Theorem 2.4.7. *Let M_{lb} be the best lower bound that can be obtained by running the algorithm of Rim and Jain on a DAG $G = (V, A)$. Further, let σ be the schedule computed by the algorithm and let $\sigma(v)$ denote the position of each $v \in V$ in σ. Let $i \in V$ be a vertex with $lb_i < \sigma(i) = ub_i^{M_{lb}}$, and let $[x, ub_i^{M_{lb}}]$ with $x \leq lb_i$ be a dense block of instructions in σ that causes i to take position $\sigma(i)$. If there exists an instruction j such that $ub_j^{M_{lb}} > ub_i^{M_{lb}}$ and $lb_j \in [x, ub_i^{M_{lb}}]$, then the earliest position of j in any feasible schedule of length M_{lb} (if it exists) is $ub_i^{M_{lb}} + 1$.*

Proof. Suppose there exists a schedule of length M_{lb} with all lower and upper bounds as given except that j's upper bound is decreased to $ub_i^{M_{lb}}$. Then, given this input, the Rim-Jain algorithm must report a lower bound less or equal to M_{lb}. However, while the constructed schedule starts equally to σ, the algorithm could now break the tie between i's and j's upper bound such that j is processed before i. In this case, j will be placed in a cycle $c \in [lb_j, ub_i^{M_{lb}}]$ with $lb_j \geq x$. This causes all the other instructions in the interval $[c, ub_i^{M_{lb}}]$ of σ (including i) to be shifted one position to the right. As a consequence, i misses its deadline and the algorithm reports a lower bound of at least $M_{lb}+1$. Hence, no schedule of length M_{lb} can exist if j is enforced to be scheduled at cycle $ub_i^{M_{lb}}$ the latest. Consequently, if a schedule of length M_{lb} does exist, j must be scheduled at cycle $ub_i^{M_{lb}}+1$ the earliest. □

2.4.3 Improving Bounds on Issue Cycles and Distances

2.4.3.1 Lower Bounds on Distances by Smart Counting

Considering regions as defined in Sect. 2.3.2.2 can help to improve the lower bound on the distance between two dependent instructions in a different way, e.g., by making use of the following lemma from [vBW01].

Lemma 2.4.8. ([vBW01]). *Let $G_{s,t} = (V_{s,t}, A_{s,t})$ be a region of a DAG $G = (V, A)$. Then it holds that $d_{s,t} \geq \min\{d_{s,v} \mid (s,v) \in A_{s,t}\} + |V_{s,t}| - 2 + \min\{d_{v,t} \mid (v,t) \in A_{s,t}\}$.*

Correctness of Lemma 2.4.8 is easy to see: After scheduling the source s of the region, the minimum latency to any successor of s must be respected before any interior vertex can be issued. Then all $|V_{s,t}| - 2$ interior vertices must be scheduled and, between the last one and the sink t, again at least the minimum latency must be respected.

Lemma 2.4.8 is a general result not specific to regions. There is a lower bound technique presented in [TC98] that exploits similar ideas. In fact, it is possible to obtain a lower bound on the distance $d_{i,k}$ between any dependent vertices $i \in V$ and $k \in V$ by determining the intermediate vertices $V_{ik} = \{j \in V \mid i \prec j \text{ and } j \prec k\}$ and computing the value $\min\{d_{i,j} \mid j \in V_{ik}\} + |V_{ik}| + \min\{d_{j,k} \mid j \in V_{ik}\}$.

Observation 2.4.9. *The lower bound on a distance $d_{i,k}$ using the above formula might be improved by ignoring some of the intermediate vertices V_{ik}.*

This is true since removing a vertex from the set V_{ik} decreases the term $|V_{ik}|$ by one but may increase one or both of the two computed minima by more than one unit. This poses a new *lower bound optimization problem*:

Problem 2.4.10. *(Distance Lower Bound Optimization Problem) Given a dependency DAG $G = (V, A)$ and two vertices $i, k \in V, i \prec k$, compute a set $V_{ik}^* \subseteq \{j \in V \mid i \prec j \text{ and } j \prec k\}$ such that $\min\{d_{i,j} \mid j \in V_{ik}^*\} + |V_{ik}^*| + \min\{d_{j,k} \mid j \in V_{ik}^*\}$ is maximum.*

In its preprocessing phase, the CP solver related to [MMvB08] either schedules regions optimally (if they are small) or applies a parameterized algorithm for the above problem to them. As far as this is known to the author, the algorithm is undocumented apart from the openly available source code, so it is briefly described here.

Consider a region $G_{s,t} = (V_{s,t}, A_{s,t})$. For ease of reference, let $W = V_{s,t} \setminus \{s,t\}$ be the set of interior vertices of the region. For $G_{s,t}$, the algorithm calculates the k smallest distances $d_{s,v_1} < d_{s,v_2} < \cdots < d_{s,v_k}$ of vertices $v \in W$ from the source. Further, it computes for each d_{s,v_i} the number $n_{s,i}$ of vertices with strictly smaller distances than d_{s,v_i}, i.e., $n_{s,i} = |\{v \in W \mid d_{s,v} < d_{s,v_i}\}|$. The same is done for the k smallest distances $d_{v_1,t} < d_{v_2,t} < \cdots < d_{v_k,t}$ of vertices $v \in W$ to the sink and the respective numbers $n_{t,i}$ of vertices with strictly smaller distances than $d_{v_i,t}$. After that, the index $i \in \{1, \ldots, k\}$ with maximum $d_{s,v_i} + |W| - n_{s,i}$ is selected. Finally, if it exists, an index $j \in \{1, \ldots, k\}$ is determined such that $|W| - n_{s,i} - n_{t,j} > 0$ and $d_{s,v_i} + d_{v_j,t} + |W| - n_{s,i} - n_{t,j}$ is maximal (cf. Fig. 2.7).

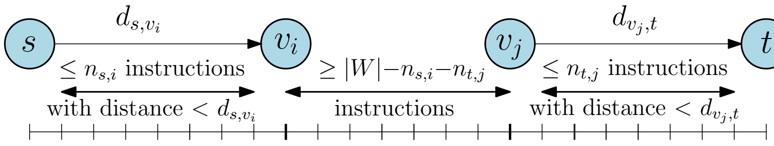

d_{s,v_i}
s
v_i
v_j
$d_{v_j,t}$
t
$\leq n_{s,i}$ instructions
with distance $< d_{s,v_i}$
$\geq |W| - n_{s,i} - n_{t,j}$
instructions
$\leq n_{t,j}$ instructions
with distance $< d_{v_j,t}$

Figure 2.7: Illustration of the different values involved in deriving a lower bound on the distance $d_{s,t}$.

The interpretation is that in the first d_{s,v_i} cycles after scheduling s at most $n_{s,i}$ instructions may be scheduled. Similarly, in the $d_{v_j,t}$ cycles immediately before scheduling t, at most $n_{t,j}$ instructions may be placed. Hence, $|W| - n_{s,i} - n_{t,j}$ is a lower bound on the number of interior vertices that can be neither scheduled earlier than d_{s,v_i} cycles after s nor later than $d_{v_j,t}$ cycles before t. If this value is smaller than zero, then it is possible that the intervals defined by s and d_{s,v_i}, and $d_{v_j,t}$ and t overlap in such a manner that summing over these two distances may not result in a valid lower bound on the distance between s and t. But, if $|W| - n_{s,i} - n_{t,j} > 0$, it is clear that there are instructions left that cannot be scheduled in any of the two intervals and, therefore, that the intervals must be disjoint (do not overlap). So in this case, at least $|W| - n_{s,i} - n_{t,j}$ instructions need to be scheduled 'in the middle' (that is, outside the two intervals) and this value can be added to the two distances in order to obtain a valid lower bound for $d_{s,t}$.

The running time of the algorithm as implemented in the CP solver can be bounded from above by $\mathcal{O}(k\ |V_{s,t}|)$. There, k is set to four. Since it is effective, a variant of this method is also incorporated into the preprocessing phase of the approach developed in Chapter 3. There, the same regions as in the CP solver are computed from the given input DAG, but regions are never solved exactly irrespective of their size. The implementation however invests $\mathcal{O}(k^2\ |V_{s,t}|)$, by testing all index pairs i and j whether they satisfy $|W| - n_{s,i} - n_{t,j} > 0$ (instead of fixing an i first) in the hope to find more regions that can be improved.

2.4.3.2 Improved Issue Cycle Bounds by Smart Counting

Since lower bounds on issue cycles of instructions are directly related to their distances from the super-source, one can use similar arguments as in the previous subsection to possibly improve them. The following techniques from [vBW01] can again be analogously applied to upper bounds on issue cycles.

Lemma 2.4.11. ([vBW01]). *Let $G = (V, A)$ be a DAG and let $v \in V$. Then for any nonempty subset $P' \subseteq P(v)$, it holds that $lb_v \geq \min\{lb_u \mid u \in P'\} + |P'| - 1 + \min\{d_{u,v} \mid u \in P'\} + 1$.*

Correctness can be easily verified by comparing Lemma 2.4.11 with Lemma 2.4.8 and replacing lb_v by $d_{b,v} + 1$, lb_u by $d_{b,u} + 1$, and then setting $s = b$ and $t = v$. Again, we may formulate an associated optimization problem.

Problem 2.4.12. *(Issue-Cycle Lower Bound Optimization Problem) Given a DAG* $G = (V, A)$ *and a vertex* $v \in V$*, compute a set* $P^* \subseteq P(v)$ *such that* $\min\{lb_u \mid u \in P^*\} + |P^*| - 1 + \min\{d_{u,v} \mid u \in P^*\} + 1$ *is maximum.*

In the solver related to [MMvB08], this problem is tackled by the following algorithm. It is quite similar to another lower bounding technique called *tighter ASAP* presented in [PJ00] but, as far as this is known to the author, undocumented apart from the openly available source code. In the following pseudocode, we assume that we may reference the lower bounds on the issue cycles and the predecessors of the vertices by using arrays.

Algorithm 3 Predecessor-based Lower Bound Algorithm

```
function PREDLB(DAG $G = (V, A)$, distances $d$, vertex $v$, lower bounds $lb$)
    SORT($P(v), lb$)                # Sort the predecessors of $v$ in increasing order of $lb$
    $pos_d \leftarrow 0$            # Position of minimum distance predecessor not yet processed
    for $i \leftarrow 1$ to $|P(v)|$ do
        if $pos_d < i$ then
            $min_d \leftarrow$ large value
            for $j \leftarrow i$ to $|P(v)|$ do
                if $min_d \geq d_{P(v)[j],v}$ then
                    $min_d \leftarrow d_{P(v)[j],v}$
                    $pos_d \leftarrow j$
        $nlb \leftarrow lb[P(v)[i]] + |P(v)| - i - 1 + min_d + 1$
        if $nlb > lb[v]$ then
            $lb[v] \leftarrow nlb$
```

The algorithm first sorts the predecessors of the vertex $v \in V$ by their lower bounds. Then, for each predecessor $p \in P(v)$, it constructs a lower bound on v's issue cycle by summing up the lower bound of p, the number of predecessors with a greater-or-equal lower bound, and the minimum distance to v among these.

In fact, Algorithm 3 can quite easily be improved by using the distances $d_{p,v}$ of predecessors $p \in P(v)$ as a second criterion to break ties such that whenever multiple predecessors of v have the same lower bound, one with the smallest distance to v will be processed first. As a result, in the next iteration, the term $|P(v)| - i - 1$ decreases by one, but the value min_d might increase by a larger amount, potentially leading to better lower bounds.

Algorithm 3 runs in time $\mathcal{O}(|P(v)|^2)$ in the worst case and the improved version (with the secondary sorting key) is used within the approaches presented in Chapter 3, too.

2.4.3.3 Improved Issue Cycle Bounds by Interval Considerations

Another method to improve lower bounds on issue cycles and to also remove symmetry from the problem can be derived by considering time intervals. Let $I(a, b)$

be the set of instructions that can possibly be scheduled in the interval $[a, b]$, i.e., $I(a, b) = \{i \mid i \in I,\ lb_i \leq b \text{ and } ub_i \geq a\}$.

Lemma 2.4.13. ([MMvB06]). *If there exists an interval $[a, b]$ such that (i) for all $i \in I(a, b)$ it holds that $ub_i = b$, (ii) for all $i \in I(a, b)$, and for all $s \in S(i)$ it holds that $lb_s - d_{i,s} - 1 \geq b$ and (iii) $|I(a, b)| \leq (b - a + 1)$, then the lower bounds lb_i of all the instructions $i \in I(a, b)$ can be set to a.*

A proof of this lemma with case distinctions can be found in [Mal08]. Exploiting symmetry conditions, it may also be formulated for upper bounds:

Lemma 2.4.14. ([MMvB06]). *If there exists an interval $[a, b]$ such that (i) for all $i \in I(a, b)$ it holds that $lb_i = a$, (ii) for all $i \in I(a, b)$ and for all $p \in P(i)$ it holds that $ub_p + d_{p,i} + 1 \leq a$, and (iii) $|I(a, b)| \leq (b - a + 1)$, then the upper bounds ub_i of all the instructions $i \in I(a, b)$ can be set to b.*

The solver related to [MMvB08] uses some fast heuristic tests to check whether there are intervals that satisfy these conditions. The corresponding routine first determines sets I_a (I_b) of instructions that satisfy condition (i) for some lower bound a (upper bound b). A suitable b (a) is then selected based on the size of the respective I_a (I_b) under consideration and it is tested whether condition (ii) is satisfied. If this is the case, it is finally tested whether there are other instructions $i \notin I_a$ ($i \notin I_b$) intersecting with the derived interval $[a, b]$. If not, the upper bounds (lower bounds) are altered to b (a) if this is an improvement. In the implementations related to the solver developed in Chapter 3, only the routine for upper bounds is used because the version for lower bounds is in conflict with the novel symmetry reduction scheme presented in Sect. 2.4.5.3.

Another useful concept in order to improve bounds based on interval considerations are so-called *Hall intervals* having their name from their relation to Philip Hall's marriage theorem proved in 1935 [Hal35].

Definition 2.4.15. (Hall interval [Hal35, Pug98, vH01]). *Let $I^*(a, b)$ be the set of instructions that can be scheduled in the interval $[a, b]$ only, i.e., $I^*(a, b) = \{i \mid i \in I, lb_i \geq a \text{ and } ub_i \leq b\}$. The interval $[a, b]$ is called a Hall interval if $|I^*(a, b)| = b - a + 1$.*

Hall intervals are those intervals where there is a known set of instructions $I^*(a, b)$ that must consume all the cycles provided by the interval $[a, b]$. It is easy to see that, if $[a, b]$ is a Hall interval and i is an instruction that is in $I(a, b)$ but not in $I^*(a, b)$, then the interval $[a, b]$ can be removed from the scheduling range of i. In particular, if $lb_i \in [a, b]$, then lb_i can be improved to $b + 1$. Similarly, if $ub_i \in [a, b]$, then ub_i can be improved to $a - 1$.

There exist several algorithms to quickly determine Hall intervals from a given set of scheduling ranges. If n is the number of ranges, the algorithms by Puget [Pug98] and Lopez-Ortiz et al. [LOQTvB03] both find all Hall intervals in time $\mathcal{O}(n \log n)$. In fact, these algorithms do not only find Hall intervals but also achieve bounds consistency (see Sect. 1.7) of an `alldifferent` constraint [vH01], i.e., they may reduce scheduling ranges accordingly. Mehlhorn and Thiel [MT00] provide an algorithm that runs in linear time plus the time needed to sort the end points of the scheduling ranges which might also lead to a completely linear time solution. The algorithm by Lopez-Ortiz et al. could theoretically also be proven to run in linear time when one may assume that the end points are representable by single memory words. It is used in the CP solver by Malik et al. [MMvB08, MMvB06] and also in the solver developed in this thesis. However, an additional run is carried out taking (weak) lower and upper bounds on the position of NOPs into account. The associated idea is that we can also treat an interval like a Hall interval if we know that the number of *instructions and NOPs* that must be placed in it is exactly equal to its size.

Another filtering algorithm that achieves bounds consistency on an `alldifferent` constraint and can also incorporate precedences into the filtering process was proposed by Bessiere et al. [BNQW11]. Following the authors, if implemented using a sophisticated data structure for disjoint sets that allows for the operations `Find` and `Union` in constant amortized time, the algorithm has a worst case running time of $\mathcal{O}(nd)$ where d is the largest upper bound present in all variable domains. Clearly, d is a number rather than an input size and $d \geq n$. The authors claim that the asymptotic running time can be reduced to $\mathcal{O}(n^2)$ using additional implementation tricks. However, already the basic algorithm appears to be hard to implement in practice. Moreover, since the precedences already have an impact on the lower and upper bounds, they are at least indirectly incorporated by the other algorithms as well.

2.4.4 Obtaining New Precedences by DAG Transformations

Heffernan and Wilken [HW05] present a set of conditions under which additional arcs (precedences) can be inserted into a DAG without altering the optimal makespan. One of their most effective transformations is based on *sub-DAG isomorphism.*

Two graphs $G = (V, E)$ and $H = (W, F)$ are *isomorphic* if $|V| = |W|$, $|E| = |F|$, and there exists a mapping $\phi : V \to W$ such that $(u, v) \in E$ holds if and only if $(\phi(u), \phi(v)) \in F$. In other words, by relabeling the vertices of one graph, we are able to obtain the other. For weighted graphs, like our dependency DAGs, we also force the weights on the mapped edges to coincide.

Theorem 2.4.1. ([HW05]). *Let $G = (V, E)$ and $H = (W, F)$ be two isomorphic sub-DAGs. Say $V = \{v_1, \dots, v_n\}$ and $W = \{w_1, \dots, w_n\}$. If G and H are such that for all $i \in \{1, \dots, n\}$*

- *v_i and w_i are independent,*
- *for each predecessor $p \in P(v_i)$, $p \notin V$, it holds that $l(p, v_i) \leq d_{p,w_i}$,*
- *for each successor $s \in S(w_i)$, $s \notin W$, it holds that $l(w_i, s) \leq d_{v_i,s}$, and*
- *for any arc (w_i, v_j), it holds that $l(w_i, v_j) \leq d_{v_j,w_i}$,*

then adding zero-latency arcs (v_i, w_i) for all $i \in \{1, \dots, n\}$ preserves the optimal schedule length of the original DAG.

The last condition of Theorem 2.4.1 may appear counter-intuitive, because it argues over arcs that 'cross' from one sub-DAG to the other which should be impossible between two DAGs to be tested for isomorphism. However, as mentioned before, the theorem refers to *induced* sub-DAGs of vertex sets of a common larger DAG. This way, there might exist such arcs between vertices V and W in the complete DAG and the last condition is then necessary in order to define a safe transformation.

Unfortunately, the detection of isomorphic sub-DAGs is $\mathcal{NP}$-complete [GJ79] (there is a simple and polynomial transformation of the general subgraph isomorphism to the sub-DAG isomorphism problem). However, in practice, a lot of small isomorphic sub-DAGs to which Theorem 2.4.1 may be applied can be found by some rather simple heuristic tests [Mal08] and the resulting search space reduction justifies the invested computation time in the constraint programming solver [MMvB08]. Hence, a similar procedure to the one that is implemented there is used within the solver implementation presented in Chapter 3, too.

2.4.5 New Ideas to Reduce the Search Space and Break Symmetries

2.4.5.1 New Precedences from Overlapping Intervals

A very simple rule to obtain a new precedence relationship that can however be applied quite frequently in practice is the following.

Theorem 2.4.16. *Let $k \in I$ be an instruction with $ub_k - lb_k = 1$. Suppose now that there exist two instructions $i, j \in I \setminus \{k\}$ such that $ub_i = ub_k$ and $lb_j = lb_k$. Then i must be a predecessor of j in any feasible schedule of I.*

Proof. The situation associated to this lemma is depicted in Fig. 2.8. We consider the two cases where instruction k might be scheduled. Suppose k is scheduled in cycle lb_k. Then instruction j can be scheduled earliest in cycle $lb_k + 1 = ub_k$. Since $ub_i = ub_k$, it follows immediately, that i must be placed before k and, therefore, also before j. In the other case, k is scheduled in cycle ub_k, it is clear that instruction i must be scheduled at cycle $ub_k - 1$ the latest. Since $ub_k - 1 = lb_k = lb_j$, it follows that i must precede j also in this case. □

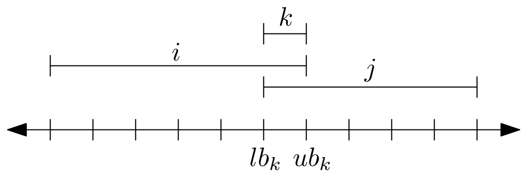


Figure 2.8: Illustration of the instructions i, j, and k of Lemma 2.4.16.

The implication is in a way similar to some of the reasonings summarized by Vilím [Vil07, Vil11]. The difference is however that Vilím considers instructions with arbitrary processing times that always have some minimum overlap irrespective of their exact starting times. Accumulating these overlaps, he derives a lower bound on the amount of resource usage in certain time intervals. These methods cannot lead to same result for the depicted situation because there cannot be any overlap of the instructions. Moreover, if an overlap was implied, then the instance would clearly be infeasible.

Tests for the condition stated by Lemma 2.4.16 can be done efficiently in practice if, for any instruction k, the instructions independent from k can be traversed in a time proportional to their number. All the instructions k with $ub_k - lb_k = 1$ can be found in linear time. For each of them, candidate pairs for i and j must be in the set of independent vertices since, after properly propagating the distances, for any successor s of k it must hold that $lb_s \geq lb_k + 1$ and for any predecessor p of k it must hold that $ub_p \leq ub_k - 1$ (cf. Sect. 2.3.3).

2.4.5.2 New Precedences and Bounds from (Hall) Intervals

Let $[a, b]$ be a Hall interval with instruction set $I^*(a, b)$.

Observation 2.4.17. *Let $p \in I \setminus I^*(a, b)$ be a predecessor of one of the instructions $i \in I^*(a, b)$ that is itself not in the interval. Then p is a predecessor of* all *the instructions in the set $I^*(a, b)$. The same is true for successors $s \in I \setminus I^*(a, b)$ of any instruction from the set $I^*(a, b)$.*

If a model has a notion of handling NOPs individually, the same observation can be made concerning NOPs. Moreover, one can restrict the number of NOPs between any two instructions $i, j \in I^*(a, b)$ to zero.

Even in the case where an interval $[a, b]$ is not a Hall interval, one can potentially derive some useful restrictions from it that can be expressed as constraints. Again, we consider the associated set of instructions $I^*(a, b)$ that must be scheduled within $[a, b]$ and the following additional sets:

- $I_{\leq} = \{v \in V \setminus I^*(a, b) : ub_v \leq b\}$
- $I_{\geq} = \{v \in V \setminus I^*(a, b) : lb_v \geq a\}$

$I_{\leq}$ is the set of instructions that need to be positioned in cycle b the latest but are not contained in $I^*(a, b)$. Analogously, $I_{\geq}$ is the set of instructions that need to be positioned at cycle a the earliest, but are not contained in $I^*(a, b)$. The idea is now to compute, for each $i \in I^*(a, b)$, an individual upper bound on the number of successors from $I_{\leq}$ and on the number of predecessors from $I_{\geq}$.

Theorem 2.4.18. *Let $[a, b]$, $I^*(a, b)$, $I_{\leq}$ and $I_{\geq}$ be given as above. Let $i \in I^*(a, b)$. The number of successors of i from the set $I_{\leq}$ can be bounded by* $\min\{(b - a + 1) - |I^*(a, b)|,\ b - lb_i,\ b - a - |S(i) \cap I^*(a, b)|\}$ *and the number of predecessors of i from the set $I_{\geq}$ can be bounded by* $\min\{(b - a + 1) - |I^*(a, b)|,\ ub_i - a,\ b - a - |P(i) \cap I^*(a, b)|\}$.

Proof. Since each $i \in I^*(a, b)$ itself needs to be placed in $[a, b]$, it is clear that any successor $s \notin I^*(a, b)$ of i with upper bound less or equal to b needs to be in the interval as well. The first bound $b - a + 1 - |I^*(a, b)|$ is valid since it is exactly the remaining number of cycles not already occupied by instructions from $I^*(a,b)$. The number of successors is trivially bounded by $b - lb_i$ since i cannot start earlier than in cycle lb_i and hence at most $b - lb_i$ cycles remain afterwards. In the third bound, the number of remaining cycles after placing i, which is $b-a$, is reduced by the number of known successors of i from the set $I^*(a,b)$. Summing up, all three bounds are valid and the smallest of them gives the tightest bound on the number of successors from $I_{\leq}$. The proof for the predecessors of i from the set $I_{\geq}$ is analogous. □

2.4.5.3 Symmetry Breaking with Latest Ready Times

A novel concept to break symmetries considers the *latest* clock cycle where an instruction *must* be ready (or already scheduled) in any case.

Definition 2.4.19. *(Latest Ready Time) Let $i \in I$ be an instruction. The value $lrt_i = \max\{ub_p + \ell(p,i) + 1 \mid p \in P(i)\}$ is called the latest ready time of i.*

The latest ready time (LRT) of an instruction is not to be mixed up with the latest release time scheduling policy in preemptive scheduling algorithms that is frequently abbreviated with LRT as well. Following Definition 2.4.19, the latest ready time of instruction i is given by the maximal sum of an upper bound of a predecessor instruction p and its latency to i plus one cycle. For artificial predecessors p (those that are not given by the instance but added a posteriori by preprocessing techniques), it is convenient to assume $\ell(p,i) = 0$. Although $\ell(p,i)$ may be only a weak lower bound on the distance between p and i in an optimum schedule, it is guaranteed that instruction i must be *ready* (or already scheduled) at time lrt_i since all predecessors are scheduled, too, and all latencies induced by data dependencies must be satisfied.

In combination with the following simple but central observation, we can exploit LRTs to reduce the search space, especially if we consider a model that captures the placements of NOPs by variables and constraints as will be the case in Chapter 3.

Observation 2.4.20. *Let σ be a schedule of the instructions I with makespan M. Let $M > |I|$ and suppose that a NOP is placed at cycle c. Let $i \in I$ be an instruction with $\sigma(i) > c$ that is however ready at c. Then altering σ by scheduling i at c leads to a schedule σ' with makespan $M' \leq M$.*

Observation 2.4.20 simply states that, at each clock cycle, it is always optimal to schedule an instruction instead of a NOP if there is at least one ready instruction at hand. This is obvious since both a NOP and an instruction cover one potential delay cycle of instructions issued earlier, but an instruction that is scheduled earlier may release further potential successors earlier whereas scheduling a NOP does not. The resulting schedule must therefore be as least as good as a schedule where a NOP is preponed w.r.t. the ready instruction. As a consequence, we may define the policy that each NOP that shall be placed before instruction i must be placed before i becomes ready, i.e., in particular before cycle lrt_i (cf. Fig. 2.9). Conversely, if we know that a NOP is placed earliest at a time later than or equal to lrt_i, then it could be fixed to be after i.

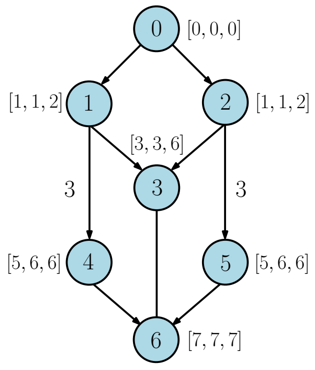


Figure 2.9: A small instance with labels $[lb_i, lrt_i, ub_i]$ at each instruction, assuming an optimal schedule length of eight cycles. Unlabeled arcs have latency zero. An optimal schedule contains a NOP either at cycle three or four. The upper bound of instruction 3 could be further reduced to four since $[5, 6]$ is a Hall interval. The LRT of instruction 3 however indicates that it is ready already at cycle three and can therefore be scheduled before the NOP.

2.5 A Novel Efficient Precedence Data Structure

Each instruction scheduler needs some data structures in order to manage predecessor and successor information. Typically, exact schedulers need to access and manipulate these data structures much more often than heuristic ones since they usually process a lot of subproblems and apply a series of search space reduction techniques. Hence, it is crucial to be able to perform the associated operations as efficiently as possible.

A classical and straightforward implementation is to have a predecessor and a successor list for each instruction. These lists allow to add a predecessor or successor in time $\mathcal{O}(1)$, and to traverse all the predecessors and successors of an instruction $i \in I$ in a time that is proportional to their number. For computational efficiency, such lists are frequently implemented using arrays. Further, they are usually complemented by another, quadratically sized, array to be able to also test in time $\mathcal{O}(1)$ whether a particular precedence exists and to retrieve the corresponding distance.

If predecessor and successor lists are realized using arrays, one would usually need to allocate two arrays, each of length $|I| - 1$, for each instruction $i \in I$. This is because, in principle, any instruction might be predecessor or successor of every other instruction. In addition, many of the discussed search space reduction techniques and as well some of the separation algorithms that will be addressed in Chapter 3 may be implemented much faster if independent instructions may be traversed in the same way as predecessors or successors. This then requires another array, leading to a total allocation of space for $3|I| - 3$ elements.

In fact, however, we know that the total space that is really used across the arrays is at most $|I| - 1$. At each point in time, for a fixed instruction $i \in I$, each instruction $j \in I \setminus \{i\}$ is either a predecessor of i, a successor of i, or independent from i.

We also know that an instruction becoming a new predecessor (successor) must be independent before. And, if an instruction j is once a predecessor (successor) of i, this relation will never change again. These simple facts and the demand for fast accesses to the independent instructions led the author to the following concept of using *nicely segmented arrays* to manage precedence information.

A nicely segmented array (NSA) is a combination of two arrays A_m and A_p. The array A_p has length $|I|-1$ and $k-1$ splitters (for the precedence relationship implementation, $k=3$). The splitters partition the array A_p into k segments. The first segment will comprise the predecessors, the second the independent instructions, and the third the successors. The first splitter sp_1 marks the end of the predecessor segment and the second splitter sp_2 marks the end of the independent instruction segment. They are to be interpreted like end-iterators in standard programming languages, i.e., they point one position behind the respective segment. Using these pointers, efficient traversal of the three segments is possible at any time by interpreting the intervals $[0, sp_1[$ as predecessors, $[sp_1, sp_2[$ as independent instructions, and $[sp_2, |I|[$ as successors. Initially, for an instruction $i \in I$, all $|I|-1$ instructions of the set $I \setminus \{i\}$ are considered to be independent. The first splitter thus points to the beginning of the array and sp_2 points one element behind it (see Fig. 2.10).

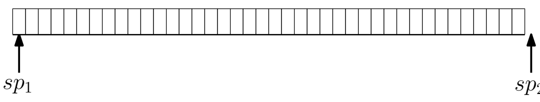


Figure 2.10: Illustration of the initial setup of the array A_p.

Adding a successor s is carried out by taking the position of the yet independent s and moving it to the end of the independent segment (if necessary). That is, the position of s will be swapped with the element at position $sp_2 - 1$. After the swap operation, the pointer sp_2 is decremented, effectively making s the first instruction of the successor area (see Fig. 2.11 for an illustration). Of course, a symmetric operation can be done for independent instructions p that shall become predecessors. Here, after swapping p with the first independent instruction, the pointer sp_1 is incremented (see Fig. 2.12). Clearly, these are constant time operations.

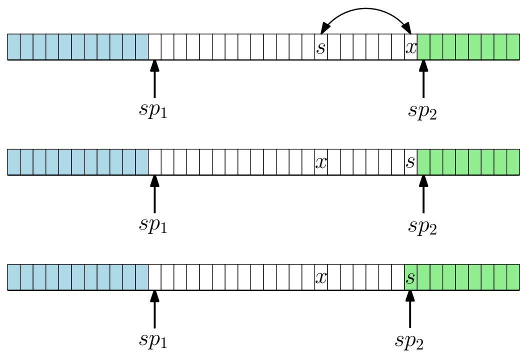


Figure 2.11: Making an independent instruction a successor instruction within the array A_p.

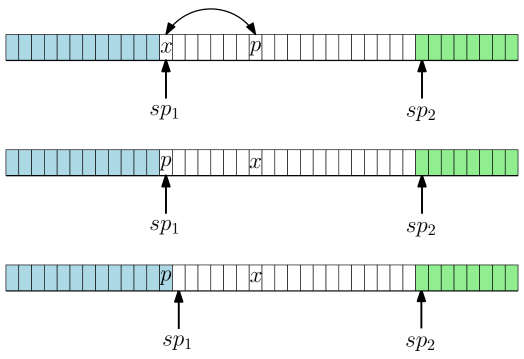


Figure 2.12: Making an independent instruction a predecessor instruction within the array A_p.

Efficient retrieval of instructions within A_p is done using the second array A_m of length $|I|$. It implements a mapping of each instruction $j \in I \setminus \{i\}$ to its position in A_p. Although the position of i never needs to be retrieved, $|I|$ rather than $|I| - 1$ elements are allocated to simplify the accesses. This way, the operations to add predecessors or successors can still be performed in constant time while acquiring space to be allocated only for $2|I| - 1$ array elements. As another feature, tests for precedence relations can also be done in time $\mathcal{O}(1)$ by a simple position comparison. A third array can be added to store the distance information, effectively making the additional quadratically sized array obsolete. Finally, by increasing k, more specialized partitionings (such as, e.g., into redundant and nonredundant predecessors/successors) are possible.

Chapter 3

Novel Integer Programming Approaches to Sequential Instruction Scheduling

While the previous chapter laid emphasis on combinatorial search space reduction techniques that exploit logical implications, this chapter focuses on the challenges that arise when mathematically modeling feasible instruction schedules. It then introduces the linear ordering problem, a well-known combinatorial optimization problem, as one possible starting point to obtain integer programming formulations of scheduling problems. After giving an overview on existing applications of the linear ordering problem in the context of scheduling, novel integer programming models to tackle the basic-block instruction scheduling problem for single-issue processors are presented. Finally, several extensions to this model are discussed and an experimental evaluation of the resulting approach is given.

3.1 The Main Challenge in Mathematically Modeling Sequential Schedules

The main challenge in modeling sequential scheduling problems mathematically is how to enforce independent instructions to attain different clock cycles. Due to the lack of a 'not equal'-relation this is a difficult task especially in linear programming.

In other words, 'not equal' means that there must be an absolute difference strictly greater than zero. So if t_i and t_j are integer variables expressing the clock cycles of two independent instructions $i, j \in I$, then it must hold that *either* $t_j \geq t_i + 1$ *or* $t_j \leq t_i - 1$. While these two inequalities are linear expressions, such *disjunctive* [BLV95] constraints are not easy to handle since, at any point in time, only one of them can be enforced and it is subject to the optimization process to find out which one. The equivalent expression $|t_j - t_i| \geq 1$ is not a linear inequality and the feasible solutions to it do not even form a convex set as is illustrated in Fig. 3.1.

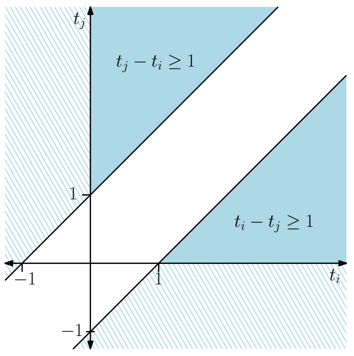


Figure 3.1: Illustration of the feasible solution spaces (shaded) associated to the inequalities $t_j \geq t_i + 1$ and $t_i \geq t_j + 1$.

Of course, it is possible to solve this problem by branching, i.e., to create two subproblems for each pair of variables where each one is equipped with exactly one of the two inequalities but this is not a promising strategy for larger instances.

One possible trick to model absolute values is to use the so-called *big-M* method. Using the mentioned integer variables t_i for the issue cycle of each instruction $i \in I$ and an additional makespan variable C_{max} (here obtaining the starting time of the last instruction) to be minimized, one could formulate the ISP as follows:

$$
\begin{array}{lllll}
\min & C_{max} & & & \\
\text{s.t.} & t_i & \leq C_{max} & \text{for all } i \in I & (3.1) \\
& t_j - t_i & \geq d_{i,j} + 1 & \text{for all } (i,j) \in R & (3.2) \\
& t_j - t_i + M b_{i,j} & \geq 1 & \text{for all } (i,j) \notin R & (3.3) \\
& t_i - t_j + M(1 - b_{i,j}) & \geq 1 & \text{for all } (i,j) \notin R & (3.4) \\
& t_i & \in \mathbb{N}_0 & \text{for all } i \in I & \\
& C_{max} & \in \mathbb{N}_0 & & \\
& b_{i,j} & \in \{0,1\} & \text{for all } (i,j) \notin R &
\end{array}
$$

The inequalities (3.1) relate the makespan variable to the largest clock cycle assigned. In a straightforward fashion, inequalities (3.2) formulate the precedence and distance constraints. Then, the condition that two independent instructions must not be assigned the same cycle is modeled by enforcing the absolute difference $|t_j - t_i| \geq 1$ via the inequalities (3.3) and (3.4). The binary variable $b_{i,j}$ controls which of the two constraints shall be active and the big-M is required to make the respective other constraint inactive. The scalar M must be chosen such that it is larger or equal to the maximum of the two maximum possible differences $t_i - t_j$ and $t_j - t_i$, i.e., if $y \geq t_j - t_i$ and $z \geq t_i - t_j$, then $M \geq \max\{y, z\}$. If $b_{i,j} = 0$, then constraint (3.3) is active and becomes $t_j - t_i \geq 1$, i.e., i will be placed before j. Further, constraint (3.4) becomes $t_i - t_j \geq 1 - M$ which is always satisfied due to the choice of M. If $b_{i,j} = 1$, then constraint (3.4) is active and constraint (3.3) is always satisfied in an analogous fashion. Applied like this, the big-M method yields an artificial convexification of the feasible region. For integer solutions, we are hence guaranteed to respect the disjunctive constraints. However, when solving the associated linear programming relaxation, we must expect to obtain solutions that are fractional in the t- and, especially, b-variables. Hence, we may be stuck in the artificially convexified area and need to branch as well. For example, for any $M \geq 2$, both constraints are satisfied when setting $t_i = t_j$ and $b_{i,j} = \frac{1}{M}$. So while this formulation works in principle, especially if strong lower and upper bounds for the variables t_i (and C_{max}) are known, it cannot be expected to be well-solvable for larger and more difficult instances.

Another common way to derive a simple formulation that does not need a big-M is to first compute an upper bound M_{ub} on the makespan such that all potentially necessary clock cycles for an optimum schedule can be expressed as the finite set $T = \{0, \ldots, M_{ub} - 1\}$. This allows for the introduction of decision variables $x_{i,t}$ for each $i \in I$ and all $t \in T$ with the meaning that:

$$x_{i,t} = \begin{cases} 1, & \text{if instruction } i \text{ is scheduled at time } t \\ 0, & \text{otherwise} \end{cases}$$

A second model that is a so-called *time-indexed formulation* (see, e.g., [Sou89]) is then:

$$\begin{array}{lllll}
\min & C_{max} & & & \\
\text{s.t.} & \sum_{t \in T} t\, x_{i,t} & \leq C_{max} & \text{for all } i \in I & (3.5) \\
& \sum_{t \in T} x_{i,t} & = 1 & \text{for all } i \in I & (3.6) \\
& \sum_{i \in I} x_{i,t} & \leq 1 & \text{for all } t \in T & (3.7) \\
& \sum_{t \in T} t\, x_{j,t} - \sum_{t \in T} t\, x_{i,t} & \geq d_{i,j} + 1 & \text{for all } (i,j) \in R & (3.8) \\
& C_{max} & \in \mathbb{N}_0 & & \\
& x_{i,t} & \in \{0,1\} & \text{for all } i \in I, \text{ for all } t \in T &
\end{array}$$

Like in the previous formulation, the first inequalities (3.5) relate the makespan variable to the largest clock cycle assigned, since $t\ x_{i,t} = t$ for exactly the clock cycle that an instruction i is assigned to, and zero otherwise. The constraints (3.6) assure that each instruction $i \in I$ is assigned exactly one clock cycle $t \in T$. Similarly, the inequalities (3.7) limit the maximum number of instructions assigned to a distinct clock cycle to at most one. Finally, inequalities (3.8) formulate the precedence and distance constraints.

Unfortunately, although this is a first viable $\{0,1\}$-IP there are again some weaknesses. First of all, one can consider time-indexed formulations to be of pseudo-polynomial size since the number of required variables is $|I| \cdot M_{ub}$ with M_{ub} being a numerical value rather than an input size. Further, the linear programming relaxations will typically split instructions over multiple clock cycles and there are several symmetric ways to do this. Nonetheless, Heffernan, Liu, and Wilken [WLH00] were able to schedule a considerable number of basic blocks using a similar model. They also found some useful cutting planes to separate fractional solutions. Still, it was shown in [vBW01] and [MMvB08] that their method is not competitive to the currently best-performing constraint programming methods.

In constraint programming models, the complexity to enforce the instructions to attain different clock cycles is usually captured using a special symbolic `alldifferent` constraint as it has been introduced in Sect. 1.7. There, it was also discussed that constraint satisfaction solvers never relax integrality such that the just mentioned issues with fractional solutions do not apply to them. CP solvers interleave branching and propagation phases. At each subproblem of the branch and bound tree, filtering algorithms for the respective constraints are used to remove infeasible values from the domains of not yet fixed variables. The filtering algorithms reach some form of local consistency, in case of the alldifferent constraint usually bounds consistency, as is also briefly covered in Sect. 1.7. The whole procedure leads to either the construction of a feasible solution or the detection that none exists with the currently fixed variable setting such that a backtracking needs to be carried out. For a more in-depth description of the concepts used in CP solvers, we refer to [vH01].

The central idea of the novel models presented in this chapter is to exploit the insight that every sequential schedule, independent whether it needs additional NOPs or not, corresponds to a certain permutation of the instructions. Even more, one may take the opposite point of view that each feasible permutation *induces* a corresponding number of NOPs that are necessary to construct a feasible schedule from it. The main aim of the newly developed formulations is to be compact in size and especially independent from the length of (an upper bound on) the optimal makespan. Last but not least, we strive for a straightforward way to model permutations of the instructions without any artificial constructions. A common approach to do this is via *linear ordering* variables. Indeed, single machine scheduling with precedences is one of the applications of the linear ordering problem (LOP) mentioned in the corresponding textbook by Martí and Reinelt [MR11]. However, as far as this is known to the author, the LOP has never been successfully and practically applied as a basis to design an integer programming formulation for single machine problems with the present form of latencies. Nevertheless, linear ordering formulations have already been proposed in the context of scheduling, especially but not exclusively, for (nondelayed) precedence-constrained single machine scheduling with and without release times and with the objective to minimize the weighted sum of completion times. In particular, for objective functions other than makespan minimization, an interesting comparison of LOP-based formulations to completion time variable and time-indexed formulations has been carried out by Keha et al. [KKF09].

3.2 The Linear Ordering Problem

3.2.1 Formal Definition

A *linear ordering* of n items $\{1, \ldots, n\}$ is a bijective function $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, i.e., equivalently a ranking, linear sequence, or permutation of the items.

Suppose that for any pair of items $i, j \in \{1, \ldots, n\}, i \neq j$, there is an associated weight (benefit) $c_{i,j}$ that becomes effective if i is ranked before j, i.e., if $\pi(i) < \pi(j)$. Then the *linear ordering problem* (LOP) on n items is the task to find an ordering π^* such that $\sum_{i=1}^{n} \sum_{j=i+1}^{n} c_{\pi^*(i),\pi^*(j)}$ is maximum. The LOP is an $\mathcal{NP}$-hard combinatorial optimization problem and has been classified as such by Garey and Johnson [GJ79]. The following descriptions are based on those in [MR11].

While there are many ways to describe the LOP, it is usually defined on a complete directed graph $G_n = (V_n, A_n)$ with arc weights $c_{i,j}$ for each $(i, j) \in A_n$. In the graph-based formulation, the LOP is to find a subset $T \subsetneq A_n$ such that (i) for every pair of vertices i and j either $(i, j) \in T$ or $(j, i) \in T$, but not both, (ii) T contains no directed cycles, and (iii) $\sum_{(i,j)\in T} c_{i,j}$ is maximum. An arc set T satisfying condition (i) is called a *tournament*, and if T additionally satisfies (ii), it is called an *acyclic tournament*.

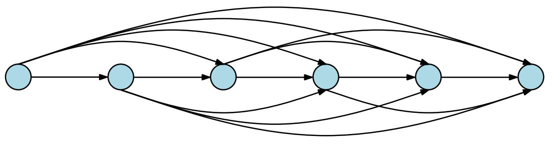


Figure 3.2: A drawing of an acyclic tournament graph with six vertices.

An acyclic tournament T, as depicted exemplarily in Fig. 3.2, can be easily identified with a linear ordering: There is only one vertex with no entering arc which corresponds to the one ranked first. The vertex with exactly one entering arc (from the first) is ranked second and so on. Interpreted like this, $\pi(i) < \pi(j)$ holds exactly for the case that $(i, j) \in T$ and vice versa. Hence, the weights $c_{i,j}$ can be directly associated with the arcs $(i, j) \in A_n$.

3.2.2 Mathematical Formulation

Relying on the graph-based interpretation, a common way to model the LOP mathematically is to define, for each arc $(i, j) \in A_n$ of $G_n = (V_n, A_n)$, a binary variable:

$$x_{i,j} = \begin{cases} 1, & \text{if } \pi(i) < \pi(j) \\ 0, & \text{otherwise} \end{cases}$$

Then, for $n \geq 3$, the LOP can be stated as an integer program as follows:

$$\begin{aligned} \max \quad & \sum_{(i,j)\in A_n} c_{i,j}x_{i,j} & & \\ \text{s.t.} \quad & x_{i,j} + x_{j,i} = 1 & & \text{for all } i,j \in V_n, i \neq j && (3.9) \\ & x_{i,j} + x_{j,k} + x_{k,i} \leq 2 & & \text{for all } i,j,k \in V_n, i < j, i < k, j \neq k && (3.10) \\ & x_{i,j} \in \{0,1\} & & \text{for all } i,j \in V_n, i \neq j && \end{aligned}$$

As requested, the objective function maximizes the total weight of the selected arcs. Constraints (3.9) enforce that for each pair of vertices $i, j \in V_n$, $i \neq j$, either the arc (i, j) or the arc (j, i) has to be part of the solution, but not both. To exclude solutions containing directed cycles of three or more vertices, the so-called *three-dicycle inequalities* (3.10) are added to the formulation. In total, the model has $n(n-1) = 2\binom{n}{2}$ variables and $2\binom{n}{3}$ nontrivial constraints.

Theorem 3.2.1. *Let $G_n = (V_n, A_n)$ be a complete directed graph and let the above integer program be formulated w.r.t. G_n. Then the set of integer feasible solutions $x \in \{0,1\}^{|A_n|}$ satisfying inequalities (3.9) and (3.10) exactly corresponds to the set of acyclic tournaments $T \subsetneq A_n$ of G_n.*

Proof. $\Leftarrow$: Let $T \subsetneq A_n$ be an acyclic tournament of G_n and let $x \in \{0,1\}^{|A_n|}$ be a vector such that $x_{i,j} = 1$ if $(i, j) \in T$ and $x_{i,j} = 0$ otherwise. Since T is a tournament, for each pair of vertices $i, j \in V_n$, $i \neq j$, either $(i, j) \in T$ or $(j, i) \in T$, but never both. Hence, the inequality $x_{i,j} + x_{j,i} = 1$ must be satisfied for each such pair of vertices. Since T is also acyclic, the inequality $x_{i,j} + x_{j,k} + x_{k,i} \leq 2$ must hold for any three-dicycle $C = \{(i, j),\ (j, k),\ (k, i) \in A_n \mid i < j,\ i < k,\ j \neq k\}$. As a conclusion, the vector x as defined is a feasible solution to the integer program.

$\Rightarrow$: Now let $x \in \{0,1\}^{|A_n|}$ be a feasible solution to the integer program w.r.t. G_n and consider the set $T = \{(i, j) \in A_n \mid x_{i,j} = 1\}$. Since x is integral and $x_{i,j} + x_{j,i} = 1$ holds for each pair of vertices $i \neq j$, T must be a tournament. Further, since the inequality $x_{i,j} + x_{j,k} + x_{k,i} \leq 2$ holds for any three-dicycle $C = \{(i, j),\ (j, k),\ (k, i) \in A_n \mid i < j,\ i < k,\ j \neq k\}$, T cannot contain any three-dicycles. In particular, the reverse dicycle to C, $C' = \{(i, k),\ (k, j),\ (j, i) \in A_n \mid i < j,\ i < k,\ j \neq k\}$, is exactly covered by the three-dicycle inequality that results from exchanging the roles of j and k, i.e., $x_{i,k} + x_{k,j} + x_{j,i} \leq 2$. It remains to show that T also does not contain any directed cycles of length $k > 3$. We prove only the case $k = 4$ since it will become apparent that the derived contradiction can be analogously constructed for a larger k (see also the argumentation in [MR11]). The proof is based on the observation that because x is integral and $x_{i,j} + x_{j,i} = 1$ holds, it must also hold that $x_{i,j} = 1 - x_{j,i}$. Suppose that $C_4 = \{(i, j), (j, k), (k, l), (l, i) \mid i < j,\ i < k,\ i < l\} \in T$ with pairwise different vertices is a directed cycle of length four. Then it follows that $x_{i,j} + x_{j,k} + x_{k,l} + x_{l,i} = 4$. However, this is a contradiction since:

$$\begin{aligned}
x_{i,j} + x_{j,k} + x_{k,l} + x_{l,i} &= x_{i,j} + x_{j,k} + (x_{k,i} - x_{k,i}) + x_{k,l} + x_{l,i} \\
&= x_{i,j} + x_{j,k} + x_{k,i} + (x_{i,k} - 1) + x_{k,l} + x_{l,i} \\
&= \underbrace{x_{i,j} + x_{j,k} + x_{k,i}}_{\leq 2} + \underbrace{x_{i,k} + x_{k,l} + x_{l,i}}_{\leq 2} - 1 \\
&\leq 4 - 1 \\
&= 3
\end{aligned}$$

□

3.2.3 Projection

The observation that $x_{i,j} + x_{j,i} = 1$ implies $x_{i,j} = 1 - x_{j,i}$ for any integral solution allows for the elimination of half of the $n(n-1)$ variables. It suffices then to define the variables

$$x_{i,j} = \begin{cases} 1, & \text{if } \pi(i) < \pi(j) \\ 0, & \text{if } \pi(j) < \pi(i) \end{cases}$$

only for $i < j$ [GJR84]. As a consequence, the three-dicycle inequalities need to be divided into two parts. The projected formulation is as follows:

$$\begin{aligned}
\max \quad & \sum_{i,j \in V_n, i<j} (c_{i,j} - c_{j,i})\, x_{i,j} \\
\text{s.t.} \quad & x_{i,j} + x_{j,k} - x_{i,k} \geq 0 && \text{for all } i,j,k \in V_n, i<j<k && (3.11) \\
& x_{i,j} + x_{j,k} - x_{i,k} \leq 1 && \text{for all } i,j,k \in V_n, i<j<k && (3.12) \\
& x_{i,j} \in \{0,1\} && \text{for all } i,j \in V_n, i<j
\end{aligned}$$

The new objective function coefficients now reflect the (positive or negative) 'gain' to set $x_{i,j} = 1$ instead of $x_{i,j} = 0$. Hence, compared to the original formulation, the objective function value differs by the constant $\sum_{i,j \in V_n, i>j} c_{j,i}$. The projected formulation has $\binom{n}{2}$ variables and $2\binom{n}{3}$ nontrivial constraints.

3.2.4 The Linear Ordering Polytope

Let $m = \binom{n}{2}$. For $n \geq 3$, the *linear ordering polytope* can be described as $P^n_{LO} = \text{conv}\{x \in \{0,1\}^m \mid x \text{ satisfies (3.11) and (3.12)}\}$ [GJR84]. The three-dicycle inequalities define facets of P^n_{LO} and completely describe it (together with the trivial inequalities) up to $n = 5$. For $n \geq 6$, many more valid and facet-inducing inequalities are known for P^n_{LO} that was intensively studied, e.g, in [GJR84, GJR85a, Fio01]. However, for many of them the associated separation problem is itself $\mathcal{NP}$-hard [MR11, Fio01] and sometimes even no practical separation algorithm is known at all. We refrain from going into further detail here, except for mentioning one of the few exceptional classes of inequalities that indeed have known polynomial-time separation procedures, namely the so-called k-fence inequalities [GJR84].

Definition 3.2.2. (k-fence inequalities [GJR84]). *Let $U = \{u_1, \ldots, u_k\}$ and $W = \{w_1, \ldots, w_k\}$ be two disjoint sets of vertices of cardinality $3 \leq k \leq \frac{|V|}{2}$. Then the inequalities*

$$\sum_{i \in \{1,\ldots,k\}} x_{u_i,w_i} + \sum_{i,j \in \{1,\ldots,k\}, i \neq j} x_{w_i,u_j} \leq k^2 - k + 1 \tag{3.13}$$

are called k-fence inequalities.

For $n \geq 6$, the k-fence inequalities define facets of P_{LO}^n [GJR85a]. They are based on particular orientations of a complete bipartite graph $K_{k,k}$, see Fig. 3.3 for an illustration of the graph corresponding to a three-fence inequality.

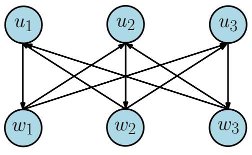


Figure 3.3: Illustration of the DAG associated with a three-fence inequality.

In the literature (e.g. [GJR84, MR11]), the three arcs x_{u_i,w_i} are sometimes called *pales* and the other six ones are called *pickets*. For each *fixed* k, an exact separation of the k-fence inequalities can be carried out in polynomial time even when doing this in an enumerative fashion. One may, e.g., enumerate over all subsets of k arcs x_{u_i,w_i} with no endpoint in common. Interpreting these k arcs as pales implies the arcs that correspond to the pickets of the associated k-fence inequality and it can easily be tested whether the inequality is violated by the current LP solution. However, even when exploiting the restriction that no two pales may share a vertex, such an approach leads to a high-order polynomial running time of $\mathcal{O}(n^{2k})$ [GJR84]. Nonetheless, the general idea to start with the pales has been proposed for heuristic separation approaches as well. The article [GJR84] describes a heuristic procedure for three-fence inequalities that is as follows. Determine three arcs x_{u_i,w_i} with no endpoint in common and such that $\frac{1}{2} - \epsilon \leq x_{u_i,w_i} \leq \frac{1}{2} + \epsilon$ for some $\epsilon \geq 0$. This methodology stems from the fact that there are basic feasible solutions to P_{LO}^n, with $x_{u_i,w_i} = \frac{1}{2}$ for all $i \in \{1,2,3\}$ and $x_{w_i,u_j} = 1$ for all $i,j \in \{1,2,3\}, i \neq j$, that violate three-fence-inequalities in a maximal way [GJR84]. As we see, a three-fence inequality can be violated by at most $\frac{1}{2}$, i.e., the left hand side can be at most 7.5. Let U and W be the ordered sets (u_1, u_2, u_3) and $W = (w_1, w_2, w_3)$ respectively. Looking at Fig. 3.3 again, we see that for some fixed U and W, the three-fence inequality associated to $U^r = (u_3, u_2, u_1)$ and $W^r = (w_3, w_2, w_1)$ has the same arcs and therefore corresponds to the same inequality. Furthermore, since there are nine arcs in total and the projection relation $x_{j,i} = 1 - x_{i.j}$ holds, the inequality associated to $U' = W$ and $W' = U$ is violated if and only if

$$\sum_{i \in \{1,\ldots,3\}} x_{u_i,w_i} + \sum_{i,j \in \{1,\ldots,3\}, i \neq j} x_{w_i,u_j} < 9 - 7 = 2.$$

So while there are six possible orders for the vertices in U for each triple of pales, we need to consider only three of them and we can perform the test for each of the resulting inequalities and their reversed counterpart in the same run.

Unfortunately, the described separation procedure could only seldom find violated three-fence inequalities on the test instances used for the evaluation in Sect. 3.7 even though ϵ was chosen to be 0.2 (whereas, in the original article, $\epsilon = 0.1$ [GJR84]). However, besides the heuristic nature of the separator, other reasons for this may be that it was only invoked if no three-dicycle inequality was violated (which is not often the case in the first iterations) and only a few cutting plane phases were carried out before the next branch takes place (see also Sect. 3.6.5).

3.3 Scheduling and Linear Ordering

Since every sequential schedule of a set of jobs corresponds to a permutation of the jobs, it is a straightforward and not a novel idea to model scheduling problems via linear ordering variables. As already stated, scheduling on a single machine with precedences is also one of the applications of the LOP mentioned in the corresponding textbook [MR11]. Perhaps more surprisingly, also the multiprocessor variant of the precedence constrained problem (without delays and with arbitrary processing times) was studied intensively in the context of linear ordering formulations by Coll et al. [CRdS06]. In general, however, it is natural to consider the problem especially in sequential contexts. The following two subsections shall give an overview of already existing applications of the LOP as well as further research related to it in the context of single-machine scheduling. Doubtless, the compiled references are by far not complete, but should allow for an impression and to find a common thread to existing literature for interested readers. Afterwards, in Sect. 3.3.3, we will discuss why the LOP is particularly well-suited to be applied to instruction scheduling.

3.3.1 Single Machine Scheduling and Linear Ordering

The most references to the LOP in terms of scheduling problems can be found in the context of single machine scheduling with (nondelayed) precedences and the objective to minimize the weighted sum of completion times, usually denoted $1|\text{prec}|\sum w_j C_j$ in the literature. There, the LOP is used as a basis for integer programming formulations, e.g., by Potts [Pot80], Boenchendorf [Boe82], Peters [Pet88], Chudak and Hochbaum [CH99], Wolsey [Wol90], Dyer and Wolsey [DW90], and Correa and Schulz [CS05]. Nemhauser and Savelsbergh [NS92] proposed a cutting plane algorithm with linear ordering variables for the same problem but minimizing the weighted sum of starting times. A different research branch that was later unified with the already mentioned one arose from Balas who proposed a model for job shop scheduling problems with (nondelayed) precedences and minimum makespan objective [Bal85]. Balas introduced disjunctive formulations that use integer variables for completion times only (similar to the first model discussed in Sect. 3.1, but for multiple machines, see also [BLV95]). Since nonpreemptive schedules are completely determined by job completion times, he studied the facial structure of the associated set of feasible solutions and coined the name *scheduling polyhedron* [Bal85]. He also outlined the relation of certain scheduling polyhedra to the linear ordering polytope whose facial structure had been investigated recently before [GJR84, GJR85a]. Balas' work on the scheduling polyhedron was complemented for single machines by Queyranne and Wang [QW91b, QW91a], and Queyranne [Que93]. The different integer programming formulations were frequently subject to the design of approximation algorithms for the objective to minimize weighted completion times. First, Hall et al. [HSSW97] derived a four-approximation algorithm based on a time-indexed formulation. Then, Schulz [Sch96], and Margot, Queyranne and Wang [MQW03], as well as Chekuri and Motwani [CM99] presented two-approximation algorithms based on linear programming relaxations with completion time variables. The LP relaxations associated to the linear ordering formulations by Potts, and Chudak and Hochbaum have an integrality gap of two and can also be used to construct two-approximation algorithms [Pot80, CH99, CM99]. For a more comprehensive survey on these results and additional references, the interested reader is kindly referred to the survey by Queyranne and Schulz [QS94]. Svensson [Sve11] showed that it is unlikely that an approximation ratio better than two can be obtained in polynomial time. Interestingly, it has also been shown that the problem $1|\text{prec}|\sum w_j C_j$ is a special case of the vertex cover problem [AM09]. Unfortunately, and although the minimization of weighted completion times is a generalization of makespan minimization, most structural results in the yet mentioned references are not very helpful in practically solving the specialized problem studied in this thesis. In fact, as already discussed in Sect. 2.4.1, there are very simple and fast combinatorial two-approximation algorithms at hand. Moreover, the addition of latency constraints introduces new difficulties that change the character of the problem considerably.

Further work that is indirectly related to the LOP considers the facial structure of the *permutahedron* (of rank n), a convex polyhedron whose extreme points are in one-to-one correspondence with the incidence vectors $x(\pi) = (\pi(1), \ldots, \pi(n)) \in \mathbb{R}^n$ of all permutations π of the set $\{1, \ldots, n\}$ [Bow72, vAFS90]. According to Ziegler [Zie95], and von Arnim, Faigle and Schrader [vAFS90], the permutahedron was introduced by Schoute already in 1911 [Sch11] and its name was coined by Guilbaud and Rosenstiehl [GR63]. It was also studied, e.g., by Gaiha and Gupta [GG77], and Young [You78]. Queyranne [Que93] showed that there is a certain bounded facet of the scheduling polyhedron that is isomorphic to the permutahedron. Bowman characterized the permutahedron using linear ordering variables already in 1972 [Bow72] (without naming them like this), and even gave a proof that, in the corresponding graph representation, there is a directed cycle of length three whenever there is one of length greater than three. Of interest is also the connection of the permutahedron to the alldifferent constraint that is discussed, e.g., in [Hoo00] and [WY01].

A very special single machine scheduling problem where pairs of jobs need to be processed such that they exactly respect a fixed distance is the *coupled task problem*. In a recent article by Békési et al. [BGJ$^+$14], an integer programming formulation based on linear ordering variables has been presented that has some similarities in handling distances to the concepts being developed in the following sections. It is well conceivable that some of the investigations made here can be translated to the coupled task problem and help to better solve the respective formulations.

3.3.2 An Interesting Relation To Parallel Machine Problems

Interestingly, a problem that is in essence very similar to instruction scheduling arises in the context of a general job shop scheduling problem with makespan objective. For each job, its responsible machine and processing time are known and there may be precedences between the jobs (that are potentially processed by different machines). Adams, Balas and Zawack [ABZ88] suggest to tackle this problem using an approach that is called the *shifting bottleneck procedure*. Effectively, the method iteratively solves a series of *one-machine sequencing problems* [Car82] to optimality.

The interesting aspect is now that the one-machine sequencing problem, though being $\mathcal{NP}$-hard, could be solved by Carlier quite effectively in practice due to a combinatorial branching rule that he could derive from heuristic schedules (called *longest tail schedules* or *Schrage schedules*). These schedules are very similar to critical path list schedules. The rule states that either the current schedule is already optimal or that a certain job j needs to be either the predecessor or successor of a certain set of *critical instructions*. This becomes even more interesting since further efforts [BLV95, DPL93] were made to identify those cases where Carlier's branching rule keeps valid even if delayed precedences come into the scene. In general, processing times cannot be treated like distances since processing times occupy a machine while distances may be covered by other instructions. However, fixing one machine conceptually, processing times of dependent jobs on other machines may be considered to impose distances between instructions on the fixed machine. Unfortunately, the preconditions necessary to still apply the branching rule when considering arbitrary delayed precedences are restrictive and were found to be seldom satisfied on the large set of test instances used for the experiments in this thesis, so that it was not further considered for the implementations. Still, this might be a pointer for other researchers that build up their models using different formulations.

3.3.3 Single-Issue Instruction Scheduling by Linear Ordering

In general, sequentially scheduling a set of jobs mainly consists of the task to determine feasible starting times for each of the jobs. The order of the jobs is then implied by their starting times. However, it is also true that, if only the order of a set of jobs has been determined, then it is often easy to compute the (minimum) corresponding feasible starting times for them based on the input data. In the case of instruction scheduling, this can, e.g., be done by an algorithm similar to the list scheduling algorithm discussed in Sect. 2.4.1.1. Assuming that the order of instructions is already given, no priority queues are necessary and only the updates of earliest starting times need to be carried out. For a dependency DAG $G = (V, A)$, the running time of such a simple algorithm can hence be reduced to $\mathcal{O}(|V| + |A|)$.

In principle, computing an ordering of instructions is therefore sufficient to also derive a schedule. However, it is not trivial to model the implications of an ordering w.r.t. its (minimum) associated makespan, i.e, the optimization over orderings, in order to truly optimize over their corresponding schedules, is a difficult challenge. The first question that arises is how to express the distance constraints. The second is how to cope with the fact that not all distance constraints can be satisfied by instructions only. In particular, a pure LOP integer program with additional distance constraints will prove infeasible as soon as NOPs are necessary to enforce a certain distance between any two instructions. Clearly, these issues cannot be resolved by simply introducing an integer variable t_i that must be larger or equal to the number of predecessors of i in the ordering, and by then enforcing distance constraints on the t-variables rather than on the LOP variables. This is because in this case, the 'all different' property needs to be additionally enforced on the t-variables which brings us back to the original problem addressed in Sect. 3.1. Indeed, the question how to model NOPs and distances led to different formulations with varying size and applicability that are presented in Sect. 3.4.

Besides these issues, there are a number of reasons why the LOP is well-suited to formulate precedence-constrained problems on single machines and, in particular, the local instruction scheduling problem for single-issue processors. First of all, employing the linear ordering variables and inequalities already solves the initially addressed problem to mathematically enforce solutions to be permutations of the instructions. Another strong advantage is that a known precedence relation $i \prec j$ can be immediately exploited by fixing the variable $x_{i,j}$ to one. Besides removing symmetry from the problem, variable fixings reduce the size of the LPs to be solved. Fortunately, due to the transitivity of precedence relationships, we may hope to be able to fix a large number of variables already based on the initial dependency DAG. Further, the search space reduction techniques presented in Sect. 2.4 will help to reduce the number of necessary variables considerably. This is also one reason why we emphasized on the derivation of new precedences there. Additionally, each branching decision on a linear ordering variable again imposes new precedences which may have a large impact and cause transitive fixings. Why this is important becomes apparent when one considers the success of the constraint programming approach by Malik et al. [MMvB08, MMvB06]. The authors manage to schedule basic blocks with more than $1,000$ instructions in a few seconds. For such instances, it is crucial that the number of (integer) variables used in their approach is linear in the number of instructions. Hence, to be able to be competitive for similar instance sizes, an integer programming formulation must make sure that the numbers of variables and constraints do not become a limitation themselves when it comes to the necessity of creating and solving (many) LPs in a few (milli)seconds. The number of variables in the approach presented here is quadratic in the number of instructions. However, it typically takes strong advantage of the reductions that are performed prior to solving the first LP. In addition, the LOP allows us to express constraints on the scheduling ranges of instructions in a straightforward manner. Similarly, restrictions on the variable sets according to the (potential) predecessors and successors of an instruction, can be explicitly specified as we will discuss in Sect. 3.4.2. A related and also covered question is how to express the notion of 'betweenness' (intermediate instructions or NOPs (to be) placed between two other dependent instructions) within the model. This can be done much better with linear ordering than with completion time or time-indexed variables. Last but not least, the LOP is a well-studied problem with profound knowledge about its polytope and the associated separation procedures for some of its facet-defining inequalities.

3.4 Novel Linear Ordering Formulations for Instruction Scheduling

3.4.1 Modeling Preliminaries

In the following, we derive new IP formulations for the instruction scheduling problem based on linear ordering variables. We directly identify the complete directed graph $G_n = (V_n, A_n)$ of the LOP with the vertices of the dependency DAG $G = (V, A)$, i.e., $V_n = V$. To ease the description, we will assume that all LOP variables w.r.t. the complete graph are present while stating additional constraints based on the arcs A of the given dependency DAG. Slightly disrespecting mathematical precision, we will not always pay attention to the fact that the variable $x_{i,j}$ only exists for $i < j$, but assume that the variable is conceptually replaced by $1 - x_{j,i}$ if $i > j$. As an example, for summing up over the predecessors of vertex j we will simply write $\sum_{i \in V_n, i \neq j} x_{i,j}$ instead of $\sum_{i \in V_n, i<j} x_{i,j} + \sum_{i \in V_n, j<i}(1 - x_{j,i})$.

3.4.2 Modeling Distances and Betweenness

For two arbitrary instructions $i, k \in I, i \neq k$, whose relative position is unknown, an instruction j is between i and k if $x_{j,k} = x_{i,j}$. This relation is not very helpful when modeling constraints as it, e.g., does not allow for a proper counting of instructions that are in between two other instructions. Fortunately, the situation is better when we already know that i precedes k. In this case, j is between i and k if and only if j is a successor of i and a predecessor of k, i.e., if $x_{i,j} x_{j,k} = 1$. This is a quadratic expression that could be linearized, but there is a preferable way to express the same information without the need for additional variables and constraints. Clearly, the product $x_{i,j} x_{j,k}$ is equal to one if and only if the sum $x_{i,j} + x_{j,k}$ is equal to two. The expression $x_{i,j} + x_{j,k} = 2$ is equivalent to $x_{j,k} + (1 - x_{j,i}) = 2$ and therefore to $x_{j,k} - x_{j,i} = 1$, stating that j is between i and k, if j is before k but not before i.

Lemma 3.4.1. *Let $G = (V, A)$ be a dependency DAG and let an instance of the LOP be defined w.r.t. G, i.e., $x_{i,k} = 1$ for all $(i, k) \in A^*$. Let x be an integral solution to the LOP. Then, for each $(i, k) \in A^*$ and each $j \in V \setminus \{i, k\}$, it holds that $x_{j,k} - x_{j,i} \geq 0$.*

Proof. Clearly, $x_{j,k} - x_{j,i} \geq -1$. So suppose that this relation holds with equality, since otherwise there is nothing to show. Then $x_{j,k} = 0$ and $x_{i,j} = 0$, and, by assumption, $x_{i,k} = 1$. Hence, the three-dicycle inequality $x_{i,j} + x_{j,k} - x_{i,k} \geq 0$ is violated by x which contradicts the assumption that x is a feasible solution to the LOP. □

For now, let us assume that the problem of modeling NOPs is absent, i.e., all required distances between two instructions could be realized with other instructions only. Based on the above relations, we are already able to express three different versions of distance constraints for dependencies $(i,k) \in A$ with distance $d_{i,k}$:

$$\sum_{j \in V \setminus \{i,k\}} x_{i,j} x_{j,k} \geq d_{i,k} \qquad \text{for all } (i,k) \in A \tag{3.14}$$

$$\sum_{j \in V \setminus \{i,k\}} (x_{i,j} + x_{j,k}) \geq 2d_{i,k} \qquad \text{for all } (i,k) \in A \tag{3.15}$$

$$\sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i}) \geq d_{i,k} \qquad \text{for all } (i,k) \in A \tag{3.16}$$

While the correctness of the inequalities (3.14) and (3.15) is obvious after the preceding discussion, validity of inequality (3.16) can be concluded from Lemma 3.4.1 and the observations made before. Clearly, the inequalities (3.15) and (3.16) are weaker than inequality (3.14) since $x_{i,j}x_{j,k} \leq \frac{1}{2}(x_{i,j} + x_{j,k})$ and the relation holds strictly as soon as at least one of $x_{i,j}$ and $x_{j,k}$ takes an LP value in $]0,1[$. However, the quadratic term prohibits us from directly using constraints (3.14) with linear programming techniques. As already stated, enforcing $x_{i,j} + x_{j,k} = 2$ is equivalent to enforcing $x_{j,k} - x_{j,i} = 1$. However, inequalities (3.16) dominate (3.15) because the latter does not account for the fact that those instructions j that are *not* between i and k also contribute to either $\sum_{j \in V \setminus \{i,k\}} x_{i,j}$ or $\sum_{j \in V \setminus \{i,k\}} x_{j,k}$.

Theorem 3.4.2. *The distance constraints (3.16) dominate the constraints (3.15).*

Proof. First, we write constraint (3.16) slightly differently:

$$\sum_{j \in V \setminus \{i,k\}} x_{j,k} - \sum_{j \in V \setminus \{i,k\}} x_{j,i} \geq d_{i,k}$$

Substituting $(1 - x_{i,j})$ for $x_{j,i}$ yields then:

$$\sum_{j \in V \setminus \{i,k\}} x_{j,k} - \sum_{j \in V \setminus \{i,k\}} (1 - x_{i,j}) \geq d_{i,k}$$

which is equal to

$$\sum_{j \in V \setminus \{i,k\}} x_{j,k} - \Big(\sum_{j \in V \setminus \{i,k\}} 1 - \sum_{j \in V \setminus \{i,k\}} x_{i,j} \Big) \geq d_{i,k}.$$

Removing the parentheses yields

$$\sum_{j \in V \setminus \{i,k\}} x_{j,k} - \sum_{j \in V \setminus \{i,k\}} 1 + \sum_{j \in V \setminus \{i,k\}} x_{i,j} \geq d_{i,k}$$

which is finally equivalent to

$$\sum_{j \in V \setminus \{i,k\}} x_{j,k} + \sum_{j \in V \setminus \{i,k\}} x_{i,j} \geq d_{i,k} + |V \setminus \{i,k\}|.$$

The left hand side of this inequality is the same as the left hand side of (3.15). Now, in order for the constraint to be feasible at all, it is necessary that $|V \setminus \{i,k\}| \geq d_{i,k}$ (in practice, it will very frequently be much larger) and so the theorem follows. $\square$

Exploiting the fact that it is impossible for an instruction to be before i and after k at the same time, we obtain a fourth way to model a distance constraint. We might equivalently state that the number of instructions that is either before i or after k is at most $|V| - d_{i,k} - 2$, i.e.:

$$\sum_{j \in V \setminus \{i,k\}} (x_{j,i} + x_{k,j}) \leq |V| - d_{i,k} - 2 \qquad \text{for all } (i,k) \in A \tag{3.17}$$

Theorem 3.4.3. *The distance constraint (3.17) is equivalent to constraint (3.16).*

Proof.

$$\begin{aligned}
&\sum_{j \in V \setminus \{i,k\}} (x_{j,i} + x_{k,j}) \leq |V| - d_{i,k} - 2 \\
\Leftrightarrow\ &\sum_{j \in V \setminus \{i,k\}} x_{j,i} + \sum_{j \in V \setminus \{i,k\}} (1 - x_{j,k}) \leq |V| - d_{i,k} - 2 \\
\Leftrightarrow\ &\sum_{j \in V \setminus \{i,k\}} x_{j,i} - \sum_{j \in V \setminus \{i,k\}} x_{j,k} \leq -d_{i,k} \\
\Leftrightarrow\ &\sum_{j \in V \setminus \{i,k\}} x_{j,k} - \sum_{j \in V \setminus \{i,k\}} x_{j,i} \geq d_{i,k}
\end{aligned}$$

$\square$

3.4.3 Lower and Upper Bound Constraints

In the following, we will frequently use the terms *lower bound constraints* and *upper bound constraints* referring to the special distance inequalities where respectively i is the super source b and k is the super sink e:

$$\sum_{j \in V \setminus \{b,k\}} x_{j,k} \geq d_{b,k} \quad \Leftrightarrow \quad \sum_{j \in V \setminus \{b,k\}} x_{j,k} - \underbrace{\sum_{j \in V \setminus \{b,k\}} x_{j,b}}_{=0} \geq d_{b,k} \tag{3.18}$$

$$\sum_{j \in V \setminus \{i,e\}} x_{i,j} \geq d_{i,e} \quad \Leftrightarrow \quad \underbrace{\sum_{j \in V \setminus \{i,e\}} x_{j,e}}_{=|V|-2} - \sum_{j \in V \setminus \{i,e\}} x_{j,i} \geq d_{i,e} \tag{3.19}$$

We call inequalities (3.18) lower bound constraints because, by the definition from Sect. 2.3.3, $lb_k = 1 + d_{b,k}$ and b itself is exactly the additional vertex preceding k in addition to those between b and k. Similarly, inequalities (3.19) are upper bound inequalities since $ub_i = M_{ub} - d_{i,e} - 1$ for any upper bound M_{ub} on the makespan and e is exactly the further subtracted vertex not between i and e.

3.4.4 Improved Distance Constraints

Let us take a closer look on the just established distance constraints

$$\sum_{j \in V \setminus \{i,k\}} x_{j,k} - \sum_{j \in V \setminus \{i,k\}} x_{j,i} \geq \; d_{i,k}.$$

The left hand side considers all the instructions $j \in V \setminus \{i, k\}$ while, except for $i = b$ and $k = e$, clearly not all these instructions are indeed candidates in order to attain a position between i and k. We will therefore aim at making the left hand side as sparse as possible such that the right hand side imposes a maximal restriction on the real candidate instructions. The following techniques will be useful not only for the formulations presented in this chapter, but for any model that uses variables permitting to express constraints on the set of instructions placed *between* two other instructions. To constitute a first simple observation, we consider lower bound constraints as special cases of distance constraints.

Observation 3.4.4. *Let $k \in V$ be an instruction with lower bound lb_k. Then there are at least lb_k cycles in the interval $[0, lb_k - 1]$ to be filled with either instructions or NOPs (see Fig. 3.4). Any instruction p that is placed at some cycle $c \in [0, lb_k - 1]$ must have itself $lb_p \leq c$ since otherwise it could not be placed there.*

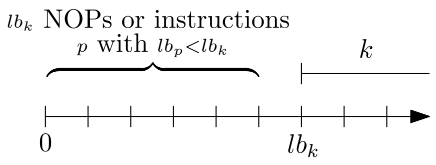


Figure 3.4: Illustration of Observation 3.4.4.

In other words, while there are potentially many more instructions (and NOPs) that might be placed before instruction k, there is only a reduced candidate set *responsible for establishing the lower bound* of k. The same is also true for upper bounds, i.e., an instruction i with upper bound ub_i must have at least $M_{ub} - ub_i - 1$ NOPs or instructions j with upper bound $ub_j > ub_i$ succeeding it. This can be exploited by restricting the variables incorporated into the lower bound constraints (3.18) to instructions from the set $J_{lb}^k = \{j \in V \setminus \{b\} \mid lb_j < lb_k\}$ and those incorporated into the upper bound constraints (3.19) to stem from the set $J_{ub}^i = \{j \in V \setminus \{e\} \mid ub_j > ub_i\}$.

A similar observation can be made and exploited for distance constraints between two instructions whose exact positions are yet unknown. Let $i, k \in V$ be a dependent pair of instructions such that $d_{i,k} > 0$. As a first step, consider the set of instructions $J = \{j \in V \setminus \{i, k\} \mid j \not\prec i,\, k \not\prec j,\, lb_j < ub_k \text{ and } ub_j > lb_i\}$. Clearly, these are all the candidates that might be between i and k, even if i and k take their respective extreme positions lb_i and ub_k. However, we may go one step further and again ask for the candidate instructions that are *responsible for establishing the distance* $d_{i,k}$ between i *and* k (in the following, we will just say *responsible*). For any position $\sigma(k)$ of k, an instruction j that is responsible can only be in the range

$[\sigma(k) - d_{i,k}, \sigma(k) - 1]$. To be conservative, we need to consider the minimal position for k which is lb_k, so the corresponding candidate set is $J_k = \{j \in V \setminus \{i,k\} \mid j \not\prec i,\, k \not\prec j,\, lb_j < ub_k$ and $ub_j \geq lb_k - d_{i,k}\}$. Similarly, for any position $\sigma(i)$ of i, instructions j responsible can only be in the range $[\sigma(i) + 1, \sigma(i) + d_{i,k}]$. Here, we need to consider the maximum possible position ub_i such that the corresponding candidate set is $J_i = \{j \in V \setminus \{i,k\} \mid j \not\prec i,\, k \not\prec j,\, lb_j \leq ub_i + d_{i,k}$ and $ub_j > lb_i\}$.

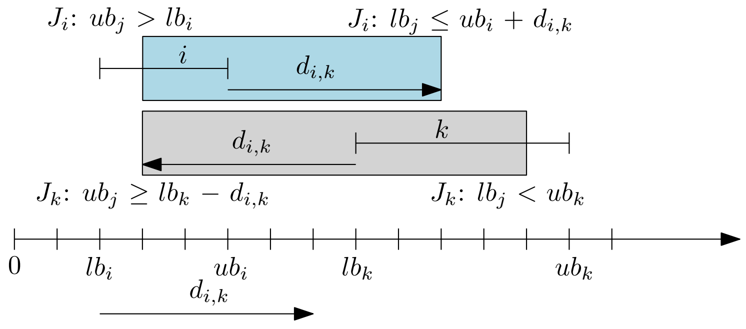


Figure 3.5: Illustration of the case for general distance relationships $d_{i,k} > 0$.

Fig. 3.5 illustrates the clock cycle intervals corresponding to J_i and J_k that must be intersected by the scheduling ranges of potentially responsible instructions. Again, the variables to be incorporated into the distance constraints can be reduced accordingly. Moreover, if the candidate sets do not coincide, then it may be beneficial to add two distance constraints related to J_i and J_k for each precedence relationship. We want to elaborate to some more extent when this is the case.

We will see that a necessary condition for the sets J_i and J_k to differ and to be smaller than J is that the distance lower bound $d_{i,k}$ must be either not binding for the lower bound of k, i.e., $lb_k > lb_i + d_{i,k}$, or not binding for the upper bound of i, i.e., $ub_i < ub_k - d_{i,k}$ (or both). Then, in general, the intersection $J_i \cap J_k$ may even be empty (this is true even in the absence of NOPs while then $|J_i \cup J_k| \geq d_{i,k}$ is a necessary condition for feasibility). Look at Fig. 3.6. By further reducing ub_i or $d_{i,k}$, or by increasing lb_k in the depicted example, the ranges for the two sets could be made completely disjoint.

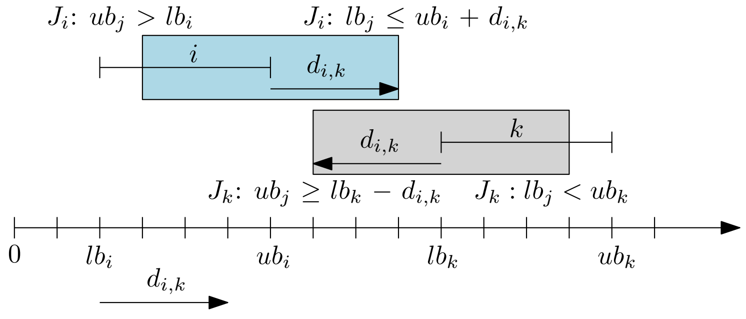


Figure 3.6: An example where there is only a small overlap of the ranges defining the sets J_i and J_k so that $|J_i \cap J_k|$ might not be large enough to cover $d_{i,k}$.

As already indicated, in the other extreme case that the distance $d_{i,k}$ is binding in both directions, i.e., $lb_k = lb_i + d_{i,k} + 1$ and $ub_i = ub_k - d_{i,k} - 1$, there will be no reduction w.r.t. the set J as defined above.

Theorem 3.4.5. *Let $i, k \in V$ be two dependent instructions such that $i \prec k$. If it holds that $lb_k = lb_i + d_{i,k} + 1$ and $ub_i = ub_k - d_{i,k} - 1$, then $J_i = J_k = J$.*

Proof. Consider the definition of $J_i = \{j \in V \setminus \{i, k\} \mid j \not\prec i, k \not\prec j, lb_j \leq ub_i + d_{i,k}$ and $ub_j > lb_i\}$. Using the second equation from the theorem, the term $lb_j \leq ub_i + d_{i,k}$ may be replaced by $lb_j \leq ub_k - 1$ which is equal to $lb_j < ub_k$ and, hence, the altered J_i matches exactly the definition of J. With the first equation, J_k can equally be turned into J. □

To close the circle to the beginning of this section, we finally remark that the lower bound constraints resemble a special case of the situation handled in Theorem 3.4.5 since for any instructions k, $k \neq b$, the distance $d_{b,k}$ is binding by definition.

Theorem 3.4.6. *Let J_b be the candidate set for b associated to the distance $d_{b,k}$ and let J^k_{lb} be the candidate set of the lower bound constraint of k. Then $J_b = J^k_{lb}$.*

Proof. Since $lb_b = ub_b = 0$ and no vertex can be a predecessor of b, we may write J_b as $J_b = \{j \in V \setminus \{b, k\} \mid k \not\prec j, lb_j \leq lb_b + d_{b,k}$ and $ub_j > 0\}$. Moreover, as $lb_b + d_{b,k} = lb_k - 1$, and $ub_j > 0$ holds for any instruction except the source, the set J_b coincides with $J^k_{lb} = \{j \in V \setminus \{b\} \mid lb_j < lb_k\}$. □

There are more special cases that allow for an exploitation during the solution process. If the positions of both vertices i and k are fixed, it might still be not clear which particular instructions are the ones to be in between. However, the distance inequality then turns into an equation since we exactly know how many instructions (and NOPs) need to be in between. In this case, all instructions that are known to be *not* in between i and k can be enforced to be either a successor or predecessor of both. Similarly, if the cardinality of a candidate set exactly matches $d_{i,k}$ and it is impossible that NOPs can occur between i and k, it is clear that exactly the instructions of the set must be those responsible for establishing the distance.

3.4.5 Modeling Idle Cycles

The just derived distance inequalities will serve as a basis to derive complete integer programming formulations for the basic-block instruction scheduling problem under the LOP model. Until here, we ignored the fact that NOPs have to be somehow incorporated into the model. Indeed, positions of instructions cannot be characterized by their rank in the permutation only and distance constraints will be infeasible if the distances cannot be established using additional variables on the left hand side.

The main challenge in modeling NOPs is that the associated variables must allow for a consistent notion of where the NOPs are placed in the schedule. This is necessary to be able to count them at all relevant places. At first sight, several possibilities come into mind. We may, for instance:

(1) Have an integer variable n_i that expresses the number of NOPs placed *immediately* before or after an instruction i.

(2) Have an integer variable n_i that expresses the *total* number of NOPs placed before or after an instruction i.

(3) Treat each NOP as an (artificial) instruction such that usual linear ordering variables are created for them.

The models that will be presented in the following are straightforward applications of these ideas. Since it is usually easy to transform the interpretations 'NOPs before' and 'NOPs after' an instruction into each other, we stick to the 'before'-variant to avoid superfluity in the following descriptions.

3.4.6 First Model with General Integer NOP Variables

When modeling NOPs based on the first concept, we introduce variables $n_i \in \mathbb{N}_0$ for all instructions $i \in V$ with the interpretation that there are exactly n_i NOPs positioned *immediately before* i.

A good property of this interpretation is that the positions of NOPs are modeled implicitly. This is in contrast to the second approach where it will be necessary to consider multiple variables in order to clarify where the NOPs are placed. However, the approach appears to be complicated in practice since the position $\sigma(i)$ of an instruction i in a schedule σ is given by the expression $\sigma(i) = n_i + \sum_{j \in V \setminus \{i\}} (1 + n_j) x_{j,i}$. The problem here is that we may account for the NOPs placed immediately before an instruction j only if j itself is before i. Hence, the term expressing the position of an instruction contains products of general integer and $\{0,1\}$-variables. This becomes also apparent in the distance inequalities that can be modeled as follows:

$$\sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i}) + \sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i}) n_j + n_k \geq d_{i,k} \qquad \text{for all } (i,k) \in A \tag{3.20}$$

Theorem 3.4.7. *Inequalities (3.20) correctly model the distance constraints.*

Proof. Following the discussion related to the distance inequality (3.16), the set of instructions $j \in J$ that are between i and k are correctly identified by the term $\sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i})$. The NOPs between i and k are therefore exactly those that are placed immediately before the instructions of the set J and immediately before k. Further, by Lemma 3.4.1, $x_{j,k} - x_{j,i} \geq 0$ must hold for all $j \in J$ due to the three-dicycle inequalities. Hence, a negative contribution of the NOP variables to the left hand side is precluded. □

The inequalities (3.20) can also be read

$$n_k + \sum_{j \in V \setminus \{i,k\}} (1 + n_j) x_{j,k} - \sum_{j \in V \setminus \{i,k\}} (1 + n_j) x_{j,i} \geq d_{i,k}.$$

Linearization of the products is possible using additional variables and constraints (cf. Sect. 1.6). Like this, we may introduce variables $\beta_{i,j} = n_i x_{i,j}$, giving the number of NOPs immediately before i if i is a predecessor of j and zero otherwise. The distance inequalities (3.20) may then be reformulated as follows:

$$\sum_{j\in V\setminus\{i,k\}}(x_{j,k} - x_{j,i}) + \sum_{j\in V\setminus\{i,k\}}(\beta_{j,k} - \beta_{j,i}) + n_k \geq \; d_{i,k} \qquad \text{for all } (i,k) \in A \tag{3.21}$$

The term $\beta_{j,k} - \beta_{j,i}$ adds the NOPs before j to the left hand side if j is before k, and subtracts them again in the case that j is also before i. Assuming N_i^{ub} to be an upper bound on the number of NOPs immediately before instruction i, a completely linearized model of the instruction scheduling problem is:

$$\begin{array}{lll}
\min & \displaystyle\sum_{i\in V} n_i & \\
\text{s.t.} & x_{i,j} + x_{j,k} - x_{i,k} \quad \geq 0 & \text{for all } i,j,k \in V, i<j<k \\
& x_{i,j} + x_{j,k} - x_{i,k} \quad \leq 1 & \text{for all } i,j,k \in V, i<j<k \\
& x_{i,k} \quad = 1 & \text{for all } (i,k) \in A^* \\
& \displaystyle\sum_{j\in V\setminus\{i,k\}}(x_{j,k} - x_{j,i} + \beta_{j,k} - \beta_{j,i}) + n_k \quad \geq d_{i,k} & \text{for all } (i,k) \in A \\
& \beta_{i,j} \quad \leq n_i & \text{for all } i,j \in V, i \neq j \\
& \beta_{i,j} - N_i^{ub} x_{i,j} \quad \leq 0 & \text{for all } i,j \in V, i \neq j \\
& \beta_{i,j} - N_i^{ub} x_{i,j} \quad \geq n_i - N_i^{ub} & \text{for all } i,j \in V, i \neq j \\
& n_i \quad \leq N_i^{ub} & \text{for all } i \in V \\
& x_{i,j} \quad \in \{0,1\} & \text{for all } i,j \in V, i<j \\
& n_i \quad \in \mathbb{N}_0 & \text{for all } i \in V \\
& \beta_{i,j} \quad \in \mathbb{N}_0 & \text{for all } i,j \in V, i \neq j
\end{array}$$

Using this model, the objective function is easy to express since the total number of NOPs is just the sum over all the NOPs that are placed immediately before each of the instructions. Besides the three-dicycle inequalities from the LOP, we have fixed variables for each precedence $(i,k) \in A^*$ and the distance inequalities as described above. The subsequent inequalities involving β-variables are all due to the linearization. The linearization is expensive in that there will be $|V|^2$ additional variables and $3|V|^2$ additional constraints. However, for many instances, a considerable number of linearizations is not necessary because the associated linear ordering variable is fixed. In this case, one may either drop the variable $\beta_{i,j}$ (if $x_{i,j} = 0$) or replace it by n_i (if $x_{i,j} = 1$) at the respective occurrences. Still, computational experiments [Tep13] revealed that the formulation leads to a relatively high number of variables in practice and has considerable problems with larger and more difficult instances. One reason for this and another drawback of the *immediate* interpretation is that the NOP variables of different instructions are unrelated to each other. This makes it difficult to apply logical implications to them and also adds a lot of symmetry to the model.

3.4.7 Second Model with General Integer NOP Variables

Following the second way to interpret NOP variables $n_i \in \mathbb{N}_0$ for all instructions $i \in V$, the variable n_i states the *total* number of NOPs placed before instruction i. Like this, the position $\sigma(i)$ of instruction i in the schedule is $\sigma(i) = n_i + \sum_{j \in V \setminus \{i\}} x_{j,i}$. We now have an important and useful relation between NOP variables in that the number of NOPs preceding k must be as least as large as the number of NOPs preceding i if k succeeds i. We first state the full model and then proceed with its description and a proof of its correctness.

$$
\begin{array}{lllll}
\min & n_e & & & \\
\text{s.t.} & x_{i,j} + x_{j,k} - x_{i,k} & \geq 0 & \text{for all } i,j,k \in V, i < j < k & \\
& x_{i,j} + x_{j,k} - x_{i,k} & \leq 1 & \text{for all } i,j,k \in V, i < j < k & \\
& x_{i,k} & = 1 & \text{for all } (i,k) \in A^* & (3.22) \\
& (n_k - n_i) + \displaystyle\sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i}) & \geq d_{i,k} & \text{for all } (i,k) \in A & (3.23) \\
& n_k & \geq n_i & \text{for all } (i,k) \in A & (3.24) \\
& n_k + M_i(1 - x_{i,k}) & \geq n_i & \text{for all } i,k \in V, i \parallel k & (3.25) \\
& n_i + M_k(1 - x_{k,i}) & \geq n_k & \text{for all } i,k \in V, i \parallel k & (3.26) \\
& x_{i,j} & \in \{0,1\} & \text{for all } i,j \in V, i < j & \\
& n_i & \in \mathbb{N}_0 & \text{for all } i \in V &
\end{array}
$$

The objective function is to minimize the number of NOPs placed before the artificial sink instruction $e \in V$. Like in the previous formulation, we have the three-dicycle inequalities from the LOP and fixed variables for each precedence $(i,k) \in A^*$. Further, for each $(i,k) \in A$, there is a distance constraint (3.23) that is composed from constraint (3.16) by adding the NOPs before k and subtracting the NOPs before i, effectively yielding the number of NOPs in between i and k. Also, for $(i,k) \in A$, we already know that $n_k \geq n_i$ (3.24) must hold. For independent instructions $i,k \in V$ however, we would usually need the following two (nonlinear) constraints in order to achieve globally consistent solutions:

$$
\begin{array}{ll}
n_k \geq \; n_i x_{i,k} & \text{for all } i,k \in V, i \parallel k \\
n_i \geq \; n_k x_{k,i} & \text{for all } i,k \in V, i \parallel k
\end{array}
$$

Here we again obtain products of a general integer and a $\{0,1\}$-variable. However, this time we do not favor linearization by the introduction of additional variables and constraints. Instead, we prefer a linearization using big-M constraints (3.25) and (3.26) in this case. Doubtless, the n-variables reintroduce the same disjunctive modeling challenges as with issue cycle variables t_i, discussed in Sect. 3.1 and Sect. 3.3.3. However, we at least move the problem from the only weakly related issue cycle variables with a large range of possible values to stronger related variables with smaller ranges. Using lower and upper bounds $N_i^{lb} \in \mathbb{N}_0$ and $N_i^{ub} \in \mathbb{N}_0$ for each of the variables n_i (that can be determined using the usual lower and upper bounds on the issue cycles of instructions), we can hope to compute relatively strong M-values. A good choice for M_i (M_k) is an upper bound on the *difference* of NOPs between k and i (i and k) in the case that k precedes (succeeds) i. Hence, M_i should be equal to $N_i^{ub} - N_k^{lb}$ (or greater) and, similarly, $M_k \geq N_k^{ub} - N_i^{lb}$ is a valid choice. In the case that k precedes i, it holds that $x_{i,k} = 0$ and the subtraction of M_i makes inequality (3.25) trivially satisfied while (3.26) is binding. The other case is analogous. For later reference, we denote the polytope corresponding to the inequalities of the integer program for a DAG $G = (V, A)$ with $v = |V|$ and $m = \binom{v}{2}$ by $P_{ISP}^G = \text{conv}\{(x, n) \in \{0, 1\}^m \times \mathbb{N}_0^v \mid x \in P_{LO}^v$ and (x, n) satisfies (3.22)-(3.26)$\}$. We will write just P_{ISP} whenever we want to refer to the set of feasible solutions of the integer program without relation to a distinct graph instance.

Theorem 3.4.8. *Let $G = (V, A)$ be a dependency DAG, $v = |V|$ and $m = \binom{v}{2}$. Then the set of integral solutions to P_{ISP}^G, i.e., the set $F_{ISP}^G = \{(x, n) \in \{0, 1\}^m \times \mathbb{N}_0^v \mid x \in P_{LO}^v$ and (x, n) satisfies* (3.22)-(3.26)$\}$ *corresponds exactly to the set of feasible schedules σ of G.*

Proof. $\Leftarrow$: Suppose a feasible schedule σ of G is given. We construct a corresponding solution $(x, n) \in F_{ISP}^G$ as follows. For each pair $i, j \in V$, $i < j$, we set $x_{i,j} = 1$ if i precedes j in σ, and $x_{i,j} = 0$ otherwise. Clearly, since σ imposes a total order on V, $x \in P_{LO}^v$. Because σ is feasible, x must also satisfy the precedence constraints (3.22). For each $i \in V$, we set n_i to the number of NOPs preceding i in σ. Hence, the position of i in σ maps exactly to $n_i + \sum_{j \in V \setminus \{i\}} x_{j,i}$. Let $(i, k) \in A^*$. By construction, $d_{i,k}^\sigma = (n_k + \sum_{j \in V \setminus \{k\}} x_{j,k}) - (n_i + \sum_{j \in V \setminus \{i\}} x_{j,i}) - x_{i,k}$. Since $x_{i,k}$ is equal to one and hence contributes only to the first sum, an equivalent expression is $d_{i,k}^\sigma = (n_k - n_i) + \sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i})$. Because σ is a feasible schedule, $d_{i,k}^\sigma \geq d_{i,k}$ and (3.23) is satisfied. For the same reason, constraints (3.24) are satisfied by the n_i variables set as described. If $x_{i,k} = 1$, constraint (3.25) coincides with (3.24) and, with the proposed choice of M_k, constraint (3.26) evaluates to $n_i \geq N_i^{lb} + n_k - N_k^{ub}$ which is trivially satisfied. If $x_{i,k} = 0$, the roles of i and k are exchanged which leads to an equally feasible situation.

$\Rightarrow$: Now suppose that $(x, n) \in F_{ISP}^G$ and we are asked to construct a feasible schedule σ of G. First, we set the position of each $i \in V$ in σ to $n_i + \sum_{j \in V \setminus \{i\}} x_{j,i}$. Since $(x, n) \in F_{ISP}^G$ implies $x \in P_{LO}^v$, x imposes a linear ordering on V. Thus, for each pair $i, k \in V$, $i \neq k$, it holds that either $\sum_{j \in V \setminus \{i\}} x_{j,i} < \sum_{j \in V \setminus \{k\}} x_{j,k}$ if $x_{i,k} = 1$, or $\sum_{j \in V \setminus \{k\}} x_{j,k} < \sum_{j \in V \setminus \{i\}} x_{j,i}$ if $x_{i,k} = 0$. Due to constraints (3.24), (3.25) and (3.26), $n_k \geq n_i$ if $x_{i,k} = 1$, or $n_i \geq n_k$ if $x_{i,k} = 0$. Summing up, either i strictly precedes k or k strictly precedes i for every pair of vertices $i, k \in V$, $i \neq k$. So each position of an instruction i in σ is unique and, due to constraint (3.23), all distances between dependent instructions are satisfied. Hence, σ is a unique feasible schedule of G. □

Since we generally aim at fixing as many linear ordering variables as possible and due to the mentioned circumstances for the NOP variables, this is a relatively elegant model. Especially, since it is rather compact in size. However, still the big-M constraints harm the strength of the model. For instance, it is possible in fractional solutions $(x, n) \in [0, 1]^m \times \mathbb{R}_0^n$ that $n_j < n_i x_{i,j}$. Another weakness is that the position of NOPs is encoded only implicitly, i.e., we cannot easily improve bounds on their positions during the optimization process and exploit them when formulating constraints. However, despite these issues, we found the model with integer NOP variables promising in our experiments.

3.4.8 Third Model with NOPs Being Artificial Instructions

Another approach is to model NOPs as what they in fact are - artificial instructions. With this interpretation, we can simply extend the graph-based LOP formulation (by setting $V = I \cup N$ with N being a set of NOP vertices) and this will already guarantee to obtain a permutation of all instructions and NOPs. The mathematical formulation of this method is then simply given by the basic LOP model extended by inequalities (3.16) and variable fixings corresponding to the precedences. We however again need an upper bound M_{ub} on the makespan in order to apply a feasible cardinality of the set N. To minimize the number of NOPs needed, one could, e.g., once more minimize the number of NOPs that are placed before the artificial sink instruction $e \in V$.

$$
\begin{array}{lllll}
\min & \sum\limits_{n \in N} x_{n,e} & & & \\
\text{s.t.} & x_{i,j} + x_{j,k} - x_{i,k} & \geq 0 & & \text{for all } i,j,k \in V, i < j < k \\
& x_{i,j} + x_{j,k} - x_{i,k} & \leq 1 & & \text{for all } i,j,k \in V, i < j < k \\
& x_{i,k} & = 1 & & \text{for all } (i,k) \in A^* \\
& \sum\limits_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i}) & \geq d_{i,k} & & \text{for all } (i,k) \in A \\
& x_{i,j} & \in \{0,1\} & & \text{for all } i,j \in V, i < j
\end{array}
$$

Besides the simplicity of the model, treating NOPs like instructions has some more advantages. First of all, the complete formulation is a $\{0,1\}$-integer program that does not need any artificial linearization at all. Further, since each NOP is an individual instance in the model, it is easy to compute a lower and upper bound on its issue cycle. These values can be incorporated into all preprocessing steps, the calculation of Hall intervals and the exploitation of logical implications at the subproblems of the branch-and-bound tree. In experiments, this has been proven to be very effective in practice. Also, a considerable number of the additional variables may be fixed in advance in order to remove symmetries from the problem. For instance, it is possible to enforce an order among the NOP vertices themselves. However, there are also disadvantages. Since the number $|N|$ of NOP vertices to add depends on (an upper bound on) the optimal schedule length (and therefore on a number rather than an input size), the size of the problem formulation is *pseudo-polynomial* and can become considerably large compared to the others, especially for instances where $|N| > |I|$.

3.4.9 A Short Comparison of the Models

As has already been indicated in the respective subsections, each of the models has its advantages and disadvantages. The first model with general integer NOP variables appeared to be computationally inferior to the others. Among the two others, there is none that is superior for all instances. If the upper bound on the number of NOPs is small, the pure LOP model from Sect. 3.4.8 is often a good choice. However, when it comes to instances with several hundreds of NOPs, the quadratic growth in the number of linear ordering variables becomes a limiting factor even if many of them may be fixed in advance. To make a decision, the model from Sect. 3.4.7 is preferred, since it remains of size $\mathcal{O}(|I|^2)$ in the number of variables and of size $\mathcal{O}(|I|^3)$ in the number of constraints, and these numbers are independent from the number of NOPs necessary to construct a feasible schedule. Still it was found appropriate to list also those models that are not in favor to be used in practice since they might trigger further ideas by other researchers. Moreover, we remark that models with a nonlinear character might become even more interesting in the future since nonlinear mixed integer programming is an emerging field at the time writing this thesis. Especially, the pure LOP model can be a good starting point to design semidefinite programming models since then the strong quadratic form (3.14) of the distance constraints can be effortlessly used.

The following classes of inequalities and the description of the developed branch-and-cut solver implementations will be presented w.r.t. the preferred model from Sect. 3.4.7. If one would like to use the pure LOP model instead, most of the constraints can be adopted by simply replacing terms with n_i-variables by terms employing a sum over the respective variables of the at most N_i^{ub} NOPs before an instruction i. To ease the description, we write, e.g., 'the number of predecessors is bounded from above by k' and mean by 'predecessors' instructions *and* NOPs, i.e., refer to the real position of an instruction within the schedule.

3.5 Additional New Classes of Inequalities

In this section, further valid inequalities for the integer program from Sect. 3.4.7 are presented. These inequalities are not necessary to obtain a complete description of feasible schedules by means of integer feasible solutions. However, they may be added to tighten the formulation or to cut off fractional LP solutions.

3.5.1 Conditional Issue Cycle Bound Constraints

For precedences $(i,k) \in A$, we know that $lb_k > lb_i$ and $ub_i < ub_k$. In contrast to that, the lower and upper bounds associated to independent pairs of vertices $i, k \in V$ are, in general, unrelated to each other. Nevertheless, a particular relative order may impose some restrictions on the positions of i and k. For instance, it is clear that k is forced to attain a position of at least $lb_i + 1$ as soon as $x_{i,k} = 1$. Similarly, i is enforced to attain a position smaller or equal to $ub_k - 1$ if $x_{i,k} = 1$. Let $lb_i > lb_k$, such that the conditional position $lb_i + 1$ is a stronger lower bound than k's usual one. The strict relation also ensures that a lower bound position of $lb_k + 1$ for k is not already implied by a strengthenend lower bound constraint and $x_{i,k} = 1$ alone. For the same reason, let $ub_k < ub_i$. Then, with $a = lb_i - lb_k$ and $b = ub_i - ub_k$, we may formulate the following inequalities:

$$n_k + \sum_{j \in V \setminus \{i,k\}} x_{j,k} \geq \; lb_k + a x_{i,k} \qquad \text{for all } i,k \in V, i \parallel k, lb_i > lb_k \qquad (3.27)$$

$$n_i + \sum_{j \in V \setminus \{i,k\}} x_{j,i} \leq \; (ub_i - 1) - b x_{i,k} \qquad \text{for all } i,k \in V, i \parallel k, ub_i > ub_k \qquad (3.28)$$

Theorem 3.5.1. *Inequalities (3.27) are valid for P_{ISP}.*

Proof. We prove the two cases for the conditional variable $x_{i,k}$ separately.

First, let $x_{i,k} = 1$. Then $\pi(i) < \pi(k)$, so the number of predecessors of k can be enforced to be at least $lb_i + 1$. In fact, the right hand side of constraint (3.27) enforces only $lb_k + lb_i - lb_k = lb_i$ predecessors. This is correct however since, in the case $x_{i,k} = 1$, i is also a predecessor that will not be accounted for on the left hand side. Now, let $x_{i,k} = 0$, i.e., $\pi(k) < \pi(i)$. Since then i is not a predecessor of k, still lb_k others need to be enforced. □

Theorem 3.5.2. *Inequalities (3.28) are valid for P_{ISP}.*

Proof. We consider again the two cases for $x_{i,k}$ separately.

If $x_{i,k} = 1$, then the number of predecessors of i is bounded by $ub_k - 1$. Clearly, k is not a predecessor of i in this case. Hence, the number of predecessors of i stemming from all other instructions (and NOPs) is bounded by $ub_i - 1 - (ub_i - ub_k) = ub_k - 1$. If $x_{i,k} = 0$, then k is a predecessor of i and only at most $ub_i - 1$ other instructions (and NOPs) may precede i. □

At this point, it should be shortly noted that the straightforward implication inequality $n_k + \sum_{j \in V \setminus \{i,k\}} x_{j,k} \geq \; lb_i x_{i,k}$ is also valid, but much weaker than inequality (3.27) for fractional values of $x_{i,k}$. In contrast to that, the upper bound version $n_i + \sum_{j \in V \setminus \{i,k\}} x_{j,i} \leq \; (ub_k - 1) x_{i,k}$ is not a valid inequality since it imposes invalid restrictions in the case that $x_{i,k} = 0$.

We now want to show that the inequalities are not only valid, but can be used as additional cutting planes for the integer program presented in Sect. 3.4.7.

Theorem 3.5.3. *P_{ISP} has fractional vertex solutions that violate inequalities (3.27), i.e., they are nonredundant.*

Proof. We prove the claim constructively by showing that there exists a basic feasible solution to the linear programming relaxation of the integer program from Sect. 3.4.7 that violates inequality (3.27). This can be done by solving the LP relaxation for a particular instance while maximizing the left hand side of inequality (3.27) in the objective function. If an optimum solution to this LP has an objective function value larger than the right hand side of (3.27), then it must correspond to a basic feasible solution that violates the inequality.

Fortunately, the instance that we will use for the proof is small and has a simple structure. It is shown in Fig. 3.7. Basically, it must only be decided which of the two orders $2-4-3$ and $3-4-2$ shall be taken. W.l.o.g., for the proof, we consider the problem as a pure feasibility problem, assuming the upper bound on the number of NOPs in the integer program to match the optimum (which is zero). Hence, the NOP variables are all fixed to zero in advance and the goal of the integer program is just to show that a feasible solution exists. The corresponding lower and upper bounds on the issue cycles of the vertices are drawn next to them in Fig. 3.7.

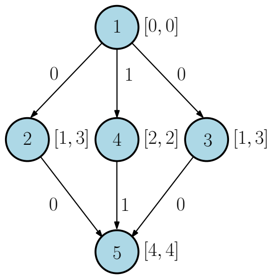


Figure 3.7: The instance used for the proofs of Theorem 3.5.3 and Theorem 3.5.8.

Let us consider vertices 2 and 3. Both have a lower bound of one and are candidates to take the places immediately before or after 4. Any vertex that is a successor of 4 must be placed at a position larger or equal to three. We choose to consider the corresponding conditional lower bound inequality (3.27) for vertex 3 with $a = lb_4 - lb_3 = 2 - 1 = 1$:

$$\begin{aligned} & n_3 + \sum_{j \in V \setminus \{3,4\}} x_{j,3} \geq lb_3 + 1x_{4,3} \\ \Leftrightarrow \quad & n_3 + x_{1,3} + x_{2,3} + x_{5,3} \geq lb_3 + x_{4,3} \end{aligned}$$

Replacing variables $x_{j,i}$ with $j > i$ by $1 - x_{i,j}$, we obtain the inequality $n_3 + x_{1,3} + x_{2,3} + x_{3,4} + (1 - x_{3,5}) \geq 2$. The only three linear ordering variables that are not fixed in advance are $x_{2,3}$, $x_{2,4}$, and $x_{3,4}$. By inserting the already fixed values, we obtain the inequality $x_{2,3} + x_{3,4} \geq 1$.

Due to its small size, we may write down the linear program completely in Fig. 3.8, omitting only the trivial inequalities and the already fixed NOP variables. The right LP in Fig. 3.8 is the reduced form of the left one after fixing also the known values of linear ordering variables.

An optimum vertex solution to this LP, that is in particular a basic feasible solution, is $x_{2,3} = 0$, $x_{2,4} = \frac{1}{2}$, and $x_{3,4} = \frac{1}{2}$. The corresponding binding inequalities are (1b), (6b), and (14) and the objective function value is $x_{2,3} + x_{3,4} = 0 + \frac{1}{2} = \frac{1}{2} < 1$. □

$$
\begin{array}{llcl}
\min & -1 + x_{1,3} + x_{2,3} + x_{3,4} - x_{3,5} & & \\
\text{s.t.} & & & \\
(1a): & x_{1,2} + x_{2,3} - x_{1,3} & \leq & 1 \\
(2a): & x_{1,2} + x_{2,4} - x_{1,4} & \leq & 1 \\
(3a): & x_{1,2} + x_{2,5} - x_{1,5} & \leq & 1 \\
(4a): & x_{1,3} + x_{3,4} - x_{1,4} & \leq & 1 \\
(5a): & x_{1,4} + x_{4,5} - x_{1,5} & \leq & 1 \\
(6a): & x_{2,3} + x_{3,4} - x_{2,4} & \leq & 1 \\
(7a): & x_{2,4} + x_{4,5} - x_{2,5} & \leq & 1 \\
(8a) & x_{3,4} + x_{4,5} - x_{3,5} & \leq & 1 \\
(1b): & -x_{1,2} - x_{2,3} + x_{1,3} & \leq & 0 \\
(2b): & -x_{1,2} - x_{2,4} + x_{1,4} & \leq & 0 \\
(3b): & -x_{1,2} - x_{2,5} + x_{1,5} & \leq & 0 \\
(4b): & -x_{1,3} - x_{3,4} + x_{1,4} & \leq & 0 \\
(5b): & -x_{1,4} - x_{4,5} + x_{1,5} & \leq & 0 \\
(6b): & -x_{2,3} - x_{3,4} + x_{2,4} & \leq & 0 \\
(7b): & -x_{2,4} - x_{4,5} + x_{2,5} & \leq & 0 \\
(8b): & -x_{3,4} - x_{4,5} + x_{3,5} & \leq & 0 \\
(9): & x_{3,2} - x_{3,1} + x_{4,2} - x_{4,1} & \geq & 0 \\
(10): & x_{2,3} - x_{2,1} + x_{4,3} - x_{4,1} & \geq & 0 \\
(11): & x_{2,4} - x_{2,1} + x_{3,4} - x_{3,1} & \geq & 1 \\
(12): & x_{3,5} - x_{3,2} + x_{4,5} - x_{4,2} & \geq & 0 \\
(13): & x_{2,5} - x_{2,3} + x_{4,5} - x_{4,3} & \geq & 0 \\
(14): & x_{2,5} - x_{2,4} + x_{3,5} - x_{3,4} & \geq & 1
\end{array}
\qquad
\begin{array}{llcl}
\min & x_{2,3} + x_{3,4} & & \\
\text{s.t.} & & & \\
 & x_{2,3} & \leq & 1 \\
 & x_{2,4} & \leq & 1 \\
 & 0 & \leq & 0 \\
 & x_{3,4} & \leq & 1 \\
 & 0 & \leq & 0 \\
 & x_{2,3} + x_{3,4} - x_{2,4} & \leq & 1 \\
 & x_{2,4} & \leq & 1 \\
 & x_{3,4} & \leq & 1 \\
 & -x_{2,3} & \leq & 0 \\
 & -x_{2,4} & \leq & 0 \\
 & 0 & \leq & 0 \\
 & -x_{3,4} & \leq & 0 \\
 & 0 & \leq & 0 \\
 & -x_{2,3} - x_{3,4} + x_{2,4} & \leq & 0 \\
 & -x_{2,4} & \leq & 0 \\
 & -x_{3,4} & \leq & 0 \\
 & -x_{2,3} - x_{2,4} & \geq & -2 \\
 & x_{2,3} - x_{3,4} & \geq & -1 \\
 & x_{2,4} + x_{3,4} & \geq & 1 \\
 & x_{2,3} + x_{2,4} & \geq & 0 \\
 & -x_{2,3} + x_{3,4} & \geq & -1 \\
 & -x_{2,4} - x_{3,4} & \geq & -1
\end{array}
$$

Figure 3.8: The linear program without trivial inequalities and NOP variables (left) and its reduced form after fixing the linear ordering variables (right).

The same LP solution also violates the conditional upper bound constraint (3.28) for vertex 2 with $b = ub_2 - ub_4 = 3 - 2 = 1$.

$$
\begin{array}{lc}
 & n_2 + \sum_{j \in V \setminus \{2,4\}} x_{j,2} \leq (ub_2 - 1) - 1x_{2,4} \\
\Leftrightarrow & n_2 + x_{1,2} + x_{3,2} + x_{5,2} \leq (ub_2 - 1) - x_{2,4} \\
\Leftrightarrow & 0 + 1 + (1 - x_{2,3}) + 0 \leq 2 - x_{2,4} \\
\Leftrightarrow & \underbrace{-x_{2,3} + x_{2,4}}_{=\frac{1}{2}} \leq 0
\end{array}
$$

Theorem 3.5.4. *P_{ISP} has fractional vertex solutions that violate inequalities (3.28), i.e., they are nonredundant.*

Lower and upper bound inequalities may be strengthened using the concepts from Sect. 3.4.4. They may be even further strengthened by taking fixed instructions and Hall intervals into account. For instance, if the conditional position $lb_i + 1$ in inequalities (3.27) is already known to be attained by another instruction (or by some indeterminate instruction associated to a Hall interval), then the conditional lower bound of k for the case $x_{i,k} = 1$ may be increased to the first cycle not already occupied.

3.5.2 Transitivity-driven Conditional Bound Constraints

Constraints similar to the usual conditional bound inequalities can be derived by considering triples of instructions $i, j, k \in V$, $i < j < k$, where *exactly one* of the three associated precedence decisions is already made. Exploiting that transitivity of precedence relationships must hold, even stronger logical implications on the bounds of instructions may be imposed using conditional expressions.

We discuss in detail the case where $i \prec j$ ($x_{i,j} = 1$) is the only decided relation. The transitivity of precedence relations (the corresponding three-dicycle inequalities) w.r.t. i, j, and k would then be violated if and only if $x_{j,k} = 1$ *and* $x_{i,k} = 0$ at the same time.

By assuming, e.g., $x_{i,k} = 0$, we can therefore conclude that $x_{j,k}$ has to be zero in any feasible schedule, too. The corresponding order is $\pi(k) < \pi(i) < \pi(j)$ so that, in this case, the position of j must be at least $lb_k + 2$ (while $lb_j \geq lb_i + 1$ already holds since $i \prec j$ is already decided). If this imposes a new constraint on j, i.e., $lb_k + 2 > lb_j$, then an inequality of the form

$$n_j + \sum_{a \in V \setminus \{j\}} x_{a,j} \geq \; lb_j + (lb_k + 2 - lb_j) x_{k,i} \tag{3.29}$$

can be added to the problem. A remarkable property of this construction is that the conditional variable $x_{k,i}$ is not related to j. Hence, the inequality imposes restrictions on the position of j from decisions made on the relative order of two other instructions. Another valid implication (where this property does not hold anymore) is that k must be placed at position $ub_j - 2$ the latest if $x_{i,k} = 0$ ($x_{k,i} = 1$). This leads to the following inequality:

$$n_k + \sum_{a \in V \setminus \{i,k\}} x_{a,k} \leq \; (ub_k - 1) - (ub_k - (ub_j - 1)) x_{k,i} \tag{3.30}$$

While the correctness of constraint (3.29) is easy to verify since it is constructed very similarly to constraint (3.27), the case of constraint (3.30) needs some formal explanation.

Theorem 3.5.5. *Inequalities (3.30) are valid for P_{ISP}.*

Proof. If $x_{k,i} = 0$, the constraint shall not be more restrictive than the usual upper bound constraint for k. In this case, i is a predecessor of k that is not counted on the left hand side, so we may enforce only at most $ub_k - 1$ other predecessors.

In the other case that $x_{k,i} = 1$, i is not a predecessor of k and the position of k shall be smaller or equal to $ub_j - 2$. Hence, it is correct to limit the number of predecessors from $V \setminus \{i, k\}$ (and preceding NOPs) by $(ub_k - 1) - (ub_k - (ub_j - 1)) = ub_k - 1 - ub_k + ub_j - 1 = ub_j - 2$. □

Similar constructions can be done by assuming $x_{j,k} = 1$. We then know that $x_{i,k}$ must also be equal to one to not violate transitivity conditions. The corresponding order is $\pi(i) < \pi(j) < \pi(k)$ so that i must be positioned at cycle $ub_k - 2$ the latest. If $ub_k - 2 < ub_i$, then we may add an inequality of the form

$$n_i + \sum_{a \in V \setminus \{i\}} x_{a,i} \leq \; ub_i - (ub_i - (ub_k - 2))x_{j,k} \tag{3.31}$$

to the problem. Since the variable $x_{j,k}$ is unrelated to i, it is not necessary to alter the right hand side like in the case of inequalities (3.30). However, it is again necessary when considering the opposite implication. The position of k must be greater or equal to $lb_i + 2$ because $x_{j,k} = 1$ means that j is a predecessor of k that is not counted on the left hand side. Hence, the corresponding inequality is:

$$n_k + \sum_{a \in V \setminus \{j,k\}} x_{a,k} \geq \; lb_k + (lb_i + 2 - lb_k - 1)x_{j,k} \tag{3.32}$$

Table 3.1 lists the complete set of possible implications when exactly one of the three precedence decisions associated to the triple $i, j, k \in V$, $i < j < k$, is known in advance. Since there are only six possible orders of the three vertices, half of listed cases are redundant and need not be considered explicitly. We discussed only the cases for $x_{i,j} = 1$ in detail since the others can be derived analogously. Strengthenings of the presented inequalities are possible in the same way as discussed for the usual conditional bound constraints.

decided	assumed	implied	order	conditional restrictions
$x_{i,j} = 1$	$x_{i,k} = 0$	$x_{j,k} = 0$	$\pi(k) < \pi(i) < \pi(j)$	$\sigma(j) \geq lb_k + 2$ and $\sigma(k) \leq ub_j - 2$
$x_{i,j} = 1$	$x_{j,k} = 1$	$x_{i,k} = 1$	$\pi(i) < \pi(j) < \pi(k)$	$\sigma(i) \leq ub_k - 2$ and $\sigma(k) \geq lb_i + 2$
$x_{i,j} = 0$	$x_{i,k} = 1$	$x_{j,k} = 1$	$\pi(j) < \pi(i) < \pi(k)$	$\sigma(j) \leq ub_k - 2$ and $\sigma(k) \geq lb_j + 2$
$x_{i,j} = 0$	$x_{j,k} = 0$	$x_{i,k} = 0$	$\pi(k) < \pi(j) < \pi(i)$	$\sigma(i) \geq lb_k + 2$ and $\sigma(k) \leq ub_i - 2$
$x_{j,k} = 1$	$x_{i,k} = 0$	$x_{i,j} = 0$	$\pi(j) < \pi(k) < \pi(i)$	$\sigma(j) \leq ub_i - 2$ and $\sigma(i) \geq lb_j + 2$
$x_{j,k} = 1$	$x_{i,j} = 1$	$x_{i,k} = 1$	$\pi(i) < \pi(j) < \pi(k)$	$\sigma(k) \geq lb_i + 2$ and $\sigma(i) \leq ub_k - 2$
$x_{j,k} = 0$	$x_{i,k} = 1$	$x_{i,j} = 1$	$\pi(i) < \pi(k) < \pi(j)$	$\sigma(j) \geq lb_i + 2$ and $\sigma(i) \leq ub_j - 2$
$x_{j,k} = 0$	$x_{i,j} = 0$	$x_{i,k} = 0$	$\pi(k) < \pi(j) < \pi(i)$	$\sigma(k) \leq ub_i - 2$ and $\sigma(i) \geq lb_k + 2$
$x_{i,k} = 1$	$x_{j,k} = 0$	$x_{i,j} = 1$	$\pi(i) < \pi(k) < \pi(j)$	$\sigma(i) \leq ub_j - 2$ and $\sigma(j) \geq lb_i + 2$
$x_{i,k} = 1$	$x_{i,j} = 0$	$x_{j,k} = 1$	$\pi(j) < \pi(i) < \pi(k)$	$\sigma(k) \geq lb_j + 2$ and $\sigma(j) \leq ub_k - 2$
$x_{i,k} = 0$	$x_{i,j} = 1$	$x_{j,k} = 0$	$\pi(k) < \pi(i) < \pi(j)$	$\sigma(k) \leq ub_j - 2$ and $\sigma(j) \geq lb_k + 2$
$x_{i,k} = 0$	$x_{j,k} = 1$	$x_{i,j} = 0$	$\pi(j) < \pi(k) < \pi(i)$	$\sigma(i) \geq lb_j + 2$ and $\sigma(j) \leq ub_i - 2$

Table 3.1: The conditional restrictions that can be obtained from considering different combinations of fixed, assumed, and implied precedence relationships.

3.5.3 Conditional NOP Constraints

Inequalities with a conditional character can also be applied w.r.t. NOPs. Let $i, k \in I$ be two independent instructions such that the lower bound on the number of NOPs before i, N_i^{lb}, is strictly larger than the corresponding lower bound N_k^{lb} of k.

The inequalities

$$n_k \geq \; N_k^{lb} + (N_i^{lb} - N_k^{lb})x_{i,k} \qquad \text{for all } i, k \in V, i \parallel k, N_i^{lb} > N_k^{lb} \tag{3.33}$$

are valid for P_{ISP} because, if $x_{i,k} = 1$, then k is a successor of i and must have at least as many preceding NOPs as i, and, if $x_{i,k} = 0$, then an inequality of this form is no more restrictive than the usual variable lower bound associated to n_k.

In addition to that, the big-M notation used in the constraints (3.25) and (3.26) allows for situations where these constraints may be violated. Let us consider inequalities (3.25) written as

$$n_k \geq\ n_i - M_i(1 - x_{i,k}) \qquad\qquad \text{for all } i, k \in V, i \parallel k$$

and assume that $x_{i,k}$ takes some fractional value. Then, for $M_i > 0$, there will be some positive amount $x = M_i(1 - x_{i,k})$ subtracted from the value of n_i on the right hand side while there will be a positive amount $y = (N_i^{lb} - N_k^{lb})x_{i,k}$ added to N_k^{lb} on the right hand side of inequalities (3.33). In particular, in any case where $n_i - x < N_k^{lb} + y$, inequalities (3.33) impose a stronger lower bound on the value of variable n_k. To construct a simple example where a solution that is binding w.r.t. the big-M constraints is violated by inequalities (3.33), assume further that $n_i = N_i^{lb}$. Then

$$\begin{aligned}
& n_i - M_i(1 - x_{i,k}) < \ N_k^{lb} + (N_i^{lb} - N_k^{lb})x_{i,k} \\
\Leftrightarrow \quad & N_i^{lb} - M_i + M_i x_{i,k} < \ N_k^{lb} + (N_i^{lb} - N_k^{lb})x_{i,k} \\
\Leftrightarrow \quad & N_k^{lb} + (N_i^{lb} - N_k^{lb}) - M_i + M_i x_{i,k} < \ N_k^{lb} + (N_i^{lb} - N_k^{lb}) - ((N_i^{lb} - N_k^{lb})(1 - x_{i,k})) \\
\Leftrightarrow \quad & -M_i + M_i x_{i,k} < \ -(N_i^{lb} - N_k^{lb}) + (N_i^{lb} - N_k^{lb})x_{i,k}
\end{aligned}$$

holds for any M_i such that $M_i > (N_i^{lb} - N_k^{lb})$. This is the usual case in practice since M_i must be chosen such that it is larger than or equal to $N_i^{ub} - N_k^{lb}$ (cf. Sect. 3.4.7). A weakness of inequalities (3.33) is however that they are only helpful in the presence of LP solutions where the variables n_i and n_k take values that are close to their lower bounds.

A symmetric version for NOP upper bounds may be formulated as well:

$$n_i \leq\ N_i^{ub} - (N_i^{ub} - N_k^{ub})x_{i,k} \qquad \text{for all } i, k \in V, i \parallel k, N_k^{ub} < N_i^{ub} \qquad (3.34)$$

3.5.4 Gap Filling cuts

The following inequalities target the frequent cases where an instruction $i \in I$ has a lower bound on its issue cycle but the set of instructions that will 'fill' these preceding cycles is not completely determined. More formally, we consider instructions $i \in I$ with issue cycle lower bound lb_i and an upper bound on the number of preceding NOPs N_i^{ub} such that $|P^*(i)| + N_i^{ub} < lb_i$. Let g be the corresponding *lower bound gap*, i.e., $g = lb_i - (|P^*(i)| + N_i^{ub})$.

By a similar argument as discussed in Sect. 3.4.4, there must be at least g instructions j currently independent from i that have a lower bound $lb_j < lb_i$. Let $I_<$ be the set of such instructions, i.e., $I_< = \{j \in I \mid j \parallel i \text{ and } lb_j < lb_i\}$ (cf. Fig. 3.9). We now change the perspective to the view that each candidate instruction $p \in I_<$ has on all the other candidate instructions. The idea is to impose a constraint on each of these instructions $p \in I_<$ based on the following simple observation.

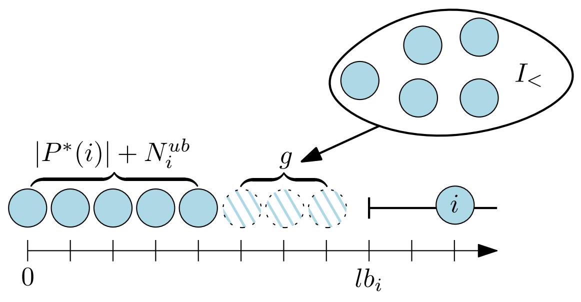


Figure 3.9: Illustration of the *lower bound gap* resulting as the difference between lb_i and the number of known predecessors $|P^*(i)|$ plus an upper bound on the number of preceding NOPs N_i^{ub} of i. The predecessors and NOPs need not necessarily be densely scheduled at the beginning although shown like this in the depicted example.

Observation 3.5.6. *Let $i \in I$ be an instruction with lower bound gap $g > 0$ and let $I_< = \{j \in I \mid j \parallel i \text{ and } lb_j < lb_i\}$. An instruction $p \in I_<$ must be a gap-filling instruction (i.e., must attain a position $\leq lb_i - 1$) if there are strictly less than g other predecessors of i from the set $I_< \setminus \{p\}$.*

In other words, we may potentially strengthen the upper bound of each $p \in I_<$ by comparing the number of i's predecessors from the set $I_< \setminus \{p\}$ with the number g of necessary ones. Whenever i has less than g predecessors from $I_< \setminus \{p\}$ closing its gap, we can conclude that p is needed to close it. This implication works only in this direction since p may still be gap-filling if i has g or more other predecessors from $I_< \setminus \{p\}$, namely whenever i attains a position strictly later than lb_i. Nonetheless, the observation can be used to construct a very effective separation scheme since we may exploit that, in *any* feasible solution and independently from p, i must have *at least* $g - 1$ predecessors from the set $I_< \setminus \{p\}$. Let $u_p = ub_p - (lb_i - 1)$ be the individual gap between the original upper bound of p and the upper bound attained if p was one of the instructions to fill the gap before lb_i. Then we may formulate the following special upper bound constraint for p:

$$n_p + \sum_{j \in I \setminus \{p\}} x_{j,p} \leq ub_p - g u_p + u_p \Big(\sum_{j \in I_< \setminus \{p\}} x_{j,i} \Big) \tag{3.35}$$

Theorem 3.5.7. *Inequalities (3.35) are valid for P_{ISP}.*

Proof. We observe first that inequality (3.35) is equivalent to the usual upper bound constraint for p if $\sum_{j \in I_< \setminus \{p\}} x_{j,i} = g$. In this case, i has exactly g and therefore sufficient predecessors from the set $I_< \setminus \{p\}$ in order to fill the lower bound gap. Thus, no stronger than its usual upper bound may be imposed on p. If $\sum_{j \in I_< \setminus \{p\}} x_{j,i} > g$, then i has even more than g predecessors from the set $I_< \setminus \{p\}$, i.e., more than necessary to fill the gap. In this case, constraint (3.35) is dominated by p's usual upper bound constraint.

The remaining and interesting case is that $\sum_{j \in I_< \setminus \{p\}} x_{j,i} < g$, i.e., there are strictly less than g predecessors from the set $I_< \setminus \{p\}$. Then p must be one of the predecessors of i responsible to close the lower bound gap. However, since we know that at least g predecessors from $I_<$ must exist, we also know that at least $g - 1$ predecessors from $I_< \setminus \{p\}$ must exist in any valid schedule. Hence, for each feasible solution, it holds that $g - \sum_{j \in I_< \setminus \{p\}} x_{j,i} \leq 1$ so that at most u_p is subtracted from p's usual lower bound ub_p. By construction, $ub_p - u_p = lb_i - 1$ as intended. □

The beneficial property of this construction is that, while we exploit that $g - \sum_{j \in I_< \setminus \{p\}} x_{j,i} \leq 1$ holds for any *integer feasible* solution, LP solutions may have a difference strictly larger than one in general. The higher the infeasibility of an LP solution is w.r.t. this relation, the stronger will be the reduction of p's upper bound, such that these solutions will frequently be cut off by the inequality. But even for fractional solutions where $0 < g - \sum_{j \in I_< \setminus \{p\}} x_{j,i} \leq 1$ holds, the inequality will lead to a (potentially scaled) reduction of p's upper bound.

We now want to substantiate more formally that the inequalities are not only valid, but can indeed be violated by fractional solutions that satisfy all the other inequalities of the integer program presented in Sect. 3.4.7.

Theorem 3.5.8. *P_{ISP} has fractional vertex solutions that violate inequalities (3.35), i.e., they are nonredundant.*

Proof. The claim can be proved using the same strategy and instance as for the nonredundancy proof of inequalities (3.27) from Sect. 3.5.1.

Vertex 4 in Fig. 3.7 has a lower bound of two and a lower bound gap g of one since its only decided predecessor is vertex 1 and no NOPs can precede it. The corresponding set $I_<$ consists of the vertices $\{2, 3\}$. For both 2 and 3, the gap u_p is equal to $3 - (2 - 1) = 2$. We choose vertex 3 as our $p \in I_<$. The corresponding inequality (3.35) for 3 is

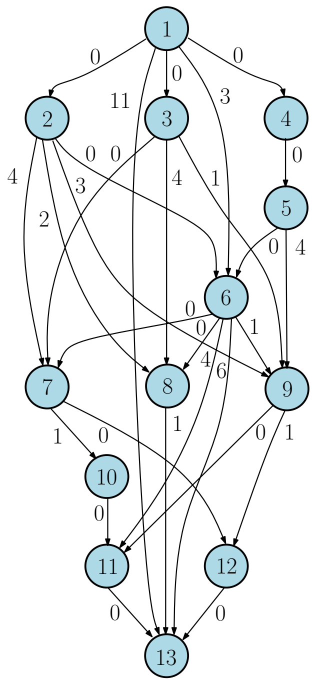
$$\begin{aligned}
& & n_3 + \sum_{j \in I \setminus \{3\}} x_{j,3} - u_3 x_{2,4} &\leq ub_3 - g u_3 \\
&\Leftrightarrow & n_3 + x_{1,3} + x_{2,3} + x_{4,3} + x_{5,3} - 2x_{2,4} &\leq 3 - 1(2) \\
&\Leftrightarrow & n_3 + x_{1,3} + x_{2,3} + (1 - x_{3,4}) + (1 - x_{3,5}) - 2x_{2,4} &\leq 1 \\
&\Leftrightarrow & n_3 + x_{1,3} + x_{2,3} - x_{3,4} - x_{3,5} - 2x_{2,4} &\leq -1
\end{aligned}$$

Inserting the already fixed values, and inverting the inequality to consider it as a minimization objective, we obtain $-1 - x_{2,3} + x_{3,4} + 2x_{2,4} \geq 0$.

Taking a close look at the reduced linear program in the right of Fig. 3.8, one can see that the value assignments $x_{2,3} = 1$, $x_{2,4} = \frac{1}{2}$, and $x_{3,4} = \frac{1}{2}$ yield another basic feasible solution with binding inequalities (1a), (6a), and (14). The corresponding objective function value is $-1 - x_{2,3} + x_{3,4} + 2x_{2,4} = -1 - 1 + \frac{1}{2} + 2 \cdot \frac{1}{2} = -\frac{1}{2} < 0$. □

We employ the LP solution from the proof to highlight some central aspects of this inequality and to give a more intuitive description of its semantics. In the associated instance, vertex 4 has a lower bound gap $g = 1$. The two candidates 2 and 3 that make up the set $I_<$ have both upper bound three. However, if one of them is used to close the lower bound gap of vertex 4, then it must attain position one (in general, a position smaller than or equal to one). The inequality for vertex 3 now states that it must be a gap-filling instruction (i.e., have its upper bound decreased to one) if less than one other vertex from $I_<$ is a predecessor of vertex 4. In this instance, the other vertex can only be the vertex 2. So suppose there is one predecessor from $I_< \setminus \{3\}$ ($x_{2,4} = 1$). Then vertex 3 is not necessarily needed to close the lower bound gap of vertex 4 and so its upper bound may not be altered. The right hand side of inequality (3.35) will then remain equal to three. However, if there is no predecessor ($x_{2,4} = 0$), then vertex 3 itself must be responsible to close the gap. Hence, its upper bound is decreased to $3 - 2(1) + 2x_{2,4} = 3 - 2 + 0 = 1$. The optimal LP solution in the proof places vertex 3 at position 2.5. The inequality however forces an upper bound of $3 - 2(1) + 2x_{2,4} = 3 - 2 + 1 = 2$. For this example, adding the inequality led the simplex algorithm to an integral solution with order $1 - 2 - 4 - 3 - 5$ when resolving the associated LP. Hence, to satisfy the inequality, $x_{2,4}$ was increased to one which in turn allows to place vertex 3 at position three again.

Theorem 3.5.9. *The polytope obtained when adding inequalities (3.27) and inequalities (3.28) to P_{ISP} has fractional vertex solutions that violate inequalities (3.35).*



v	lb_v	ub_v
1	0	0
2	1	3
3	1	5
4	1	3
5	2	4
6	4	5
7	6	8
8	6	10
9	7	9
10	8	10
11	9	11
12	9	11
13	12	12

Figure 3.10: The instance used for the proof of Theorem 3.5.9.

Proof. To prove Theorem 3.5.9, it is necessary to show the existence of a basic feasible solution to the linear programming relaxation that violates inequalities (3.35) but satisfies all inequalities (3.27) and inequalities (3.28). Such a situation is not easy to find for instances as small as the one used in the previous nonredundancy proof. A relatively small instance that could be retrieved investing reasonable time has 13 vertices and can also be scheduled optimally without NOPs. Its size permits to print the instance in Fig. 3.10, however, even the corresponding reduced LP (after removing fixed variables) has too many constraints to be completely displayed in a sensible way. We therefore refrain from that but, for the sake of reproducibility, give the LP values of all nonfixed variables and the corresponding binding inequalities in Fig. 3.11. We select vertex 8 as our target for the inequality. Vertices 8 and 9 together build up the set $I_<$ of vertex 10 that has a lower bound gap of one. Further, vertex 8 has $u_8 = ub_8 - (lb_{10} - 1) = 10 - 7 = 3$. Therefore inequality (3.35) for vertex 8 is:

$$n_8 + \sum_{j \in I \setminus \{8\}} x_{j,8} - u_8 \Big(\sum_{j \in I_< \setminus \{8\}} x_{j,10} \Big) \leq ub_8 - gu_8$$

Replacing again all variables $x_{j,i}$ with $j > i$ by $1 - x_{i,j}$, plugging in all known values and fixed variables, and multiplying the inequality with -1, we obtain:

$$-x_{7,8} + x_{8,9} + x_{8,10} + x_{8,11} + x_{8,12} + 3x_{9,10} \geq 3$$

In the basic feasible solution shown in Fig. 3.11, the left hand side of this inequality is only 2.5 and, therefore, the inequality is violated.

$$\begin{array}{rlrcr}
x_{2,3} = 1 & & -x_{2,3} - x_{2,4} - x_{2,5} & \geq & -3 \\
x_{2,4} = 1 & & x_{8,11} + x_{8,12} & \geq & 1 \\
x_{2,5} = 1 & & x_{8,10} + x_{9,10} & \geq & 1 \\
x_{3,4} = 0 & & x_{8,10} + x_{9,10} - x_{10,12} & \geq & 0 \\
x_{3,5} = 0 & & x_{2,3} + x_{2,4} + x_{2,5} - x_{7,8} - x_{7,9} & \geq & 1 \\
x_{3,6} = 1 & & -x_{2,3} + x_{3,4} + x_{3,5} + x_{3,6} + x_{7,8} - x_{8,9} - x_{8,10} - x_{8,11} - x_{8,12} & \geq & -1 \\
x_{7,8} = 1 & & -x_{2,4} + x_{2,5} - x_{3,4} + x_{3,5} & \geq & 0 \\
x_{7,9} = 1 & & -x_{2,5} - x_{3,5} + x_{7,9} + x_{8,9} - x_{9,10} & \geq & 0 \\
x_{8,9} = \frac{1}{2} & & -x_{3,6} - x_{7,8} - x_{7,9} & \geq & -3 \\
x_{8,10} = \frac{1}{2} & & -x_{3,6} + x_{8,11} - x_{11,12} & \geq & -1 \\
x_{8,11} = \frac{1}{2} & & x_{7,8} + x_{7,9} + x_{8,10} + x_{9,10} - x_{10,12} & \geq & 2 \\
x_{8,12} = \frac{1}{2} & & x_{8,11} + x_{8,12} & \geq & 1 \\
x_{9,10} = \frac{1}{2} & & x_{8,10} - x_{8,12} + x_{10,12} & \leq & 1 \\
x_{10,12} = 1 & & x_{8,9} - x_{8,11} & \leq & 0 \\
x_{11,12} = \frac{1}{2} & & x_{8,10} - x_{8,11} & \leq & 0
\end{array}$$

Figure 3.11: A basic feasible solution characterized by its associated LP values (left) and binding inequalities (right).

□

A straightforward separation procedure for these constraints can be implemented to have an asymptotic running time of $\mathcal{O}(|I|^2)$. For each instruction, the set of candidate independent instructions $I_<$ needs to be determined which takes $\mathcal{O}(|I|)$ time in the worst case. However, it is possible to derive the value $X = \sum_{j \in I_<} x_{j,i}$ during the same run. Now, for each candidate instruction $p \in I_<$ considered, computing the value $u_p(\sum_{j \in I_< \setminus \{p\}} x_{j,i})$ is possible in $\mathcal{O}(1)$ time since this is simply $u_p(X - x_{p,i})$. Having summed up $\sum_{j \in I \setminus \{p\}} x_{j,p} + n_p$ for each p in a preprocessing step (which can clearly be done in time $\mathcal{O}(|I|^2)$), violation of the inequality can be tested by a simple comparison.

The inequalities (3.35) may even be slightly strengthened and the separation procedure can be improved to potentially find more violations. So far, the property to be a predecessor (successor) of i is used as a certificate for instructions to be (not) gap-filling. However, there are other possible certificates. For instance, an instruction p is also proven (not) to be gap-filling if p is a predecessor (successor) of *any* other instruction that has a lower bound greater than or equal to lb_i. Let $L = \{r \in I \mid lb_r \geq lb_i\}$. Then we may determine, for each $j \in I_< \setminus \{p\}$, the minimal $x_{j,r}$ such that $r \in L$ and sum up over all these values instead of just over all $x_{j,i}$.

An analogous version of the constraint can be formulated and separated for *upper bound gaps* of an instruction i. In this case, potential successor candidates $I_>$ of i are determined and any $s \in I_>$ must have its lower bound increased to $ub_i + 1$ if not enough other potential successors are in fact successors of i. The associated inequality is as follows:

$$n_s + \sum_{j \in I \setminus \{s\}} x_{j,s} \geq lb_s + gl_s - l_s \Big(\sum_{j \in I_> \setminus \{s\}} x_{i,j} \Big) \tag{3.36}$$

3.5.5 Interval Filling Cuts

A similar construction as presented in Sect. 3.5.4 can be established also for intervals. Basically, this is a generalization since the whole concept can be applied as soon as there is a certain range of cycles where

- it is known how many instructions must be in it (or there is at least a good lower bound on this number),
- and there is some useful certificate that characterizes instructions placed or *not* placed in these cycles and that can be expressed by the variables at hand.

Let us consider an interval $[a, b]$, $a < b$, of clock cycles. Like in Sect. 2.4.3.3, we are interested in the sets $I^*(a, b) = \{i \in I \mid lb_i \geq a \text{ and } ub_i \leq b\}$ of mandatory and $I(a, b) = \{i \in I \mid lb_i \leq b \text{ and } ub_i \geq a\}$ of potential instructions placed in $[a, b]$.

We know that $[a, b]$ comprises $k = b - a + 1$ clock cycles. Suppose that we have an upper bound N_{ub} on the number of NOPs that could be potentially placed in $[a, b]$. Then $k' = k - N_{ub}$ is a lower bound on the number of instructions that must be placed in $[a, b]$. Assume further that $|I^*(a, b)| < k'$, i.e., the set of instructions that must fill the k' cycles is not entirely known. Consider now the set $I(a, b)$. By a similar argument as in Sect. 3.5.4, an instruction $i \in I(a, b)$ must be one of the k' interval-filling instructions if less than k' other instructions from $I(a, b) \setminus \{i\}$ are interval-filling. Let $ub_i > b$ for some $i \in I(a, b)$ and $u_i = ub_i - b$ be the difference or gap between the upper bound of i and b. Similarly, for some $i \in I(a, b)$ with $lb_i > a$, let $l_i = lb_i - a$ be the difference or gap between the lower bound of i and a. We can strengthen the upper (lower) bound of i to b (a) whenever strictly less than k' instructions from $I(a, b) \setminus \{i\}$ are in $[a, b]$. So if i_b is an instruction with upper bound b and i_a an instruction with lower bound a, then the following inequalities are valid for each instruction $i \in I(a,b) \setminus \{i_a, i_b\}$:

$$n_i + \sum_{j \in I \setminus \{i\}} x_{j,i} \leq ub_i + u_i(|I(a,b)| - k' - 1) - u_i\Big(\sum_{j \in I(a,b) \setminus \{i, i_b\}} x_{i_b,j} + \sum_{j \in I(a,b) \setminus \{i, i_a\}} x_{j,i_a}\Big) \tag{3.37}$$

$$n_i + \sum_{j \in I \setminus \{i\}} x_{j,i} \geq lb_i - l_i(|I(a,b)| - k' - 1) + l_i\Big(\sum_{j \in I(a,b) \setminus \{i, i_b\}} x_{i_b,j} + \sum_{j \in I(a,b) \setminus \{i, i_a\}} x_{j,i_a}\Big) \tag{3.38}$$

The inequalities have the same character as the gap filling cuts, except that the way of counting is reversed. Instead of counting the instructions from $I(a,b) \setminus \{i\}$ that are in $[a,b]$, we count those that are not in $[a,b]$. So in inequality (3.37), we first add $u_i(|I(a,b)| - k' - 1)$ to i's upper bound because a reduction of the upper bound shall take place only if strictly less than k' instructions from $I(a,b) \setminus \{i\}$ are in $[a,b]$ which is equivalent to the condition that strictly more than $|I(a,b) \setminus \{i\}| - k' = |I(a,b)| - k' - 1$ instructions are not in $[a,b]$. The construction for inequality (3.38) is analogous. The reversed way of counting simplifies to highlight that we can use two certificates for an instruction to be not interval-filling at a time, potentially improving the impact of the inequalities for fractional LP solutions. Each instruction $i \in I(a,b)$ is certainly *not* contributing to the k' necessary instructions if it is either a successor of any instruction i_b with $lb_i \geq b$ or a predecessor of any instruction i_a with $ub_i \leq a$. It is feasible to sum up both x_{j,i_a} and $x_{i_b,j}$ for a particular $j \in I(a,b) \setminus \{i, i_a, i_b\}$ because, in any feasible solution, j cannot be before i_a and after i_b at the same time.

Using the same strengthening principle as in Sect. 3.5.4, the inequalities and also the separation procedure may be improved by selecting, for each $j \in I(a,b)$, the maximal $x_{i_b,j}$ (x_{j,i_a}) of *all* instructions $i_b \neq i, j$ with upper bound b ($i_a \neq i, j$ with lower bound a).

Again, we consider a small example to show the benefits of these constructions. Consider a concrete interval $[52, 65]$ with $k = 14$. Suppose that at most two NOPs might be in $[52, 65]$, so that at least $k' = 12$ instructions must be placed in the interval. Further, suppose that $|I(52, 65)| = 20$, and that there is an instruction $i \in I(52, 65)$ with scheduling range $[lb_i, 67]$, i.e., $u_i = 2$. Now, let us take a look at the right hand side of constraint (3.37) for i. Inserting all the known values of this example, the right hand side limits the number of predecessors of i to $67 + 2(20 - 12 - 1) - 2(X)$ where X is equal to a number of instructions from $I(52, 65) \setminus \{i\}$ being definitely outside the interval. Twelve of the 20 instructions in $I(52, 65)$ must be in the interval, hence at most eight might be out of the interval. For $X \leq 7$, it would therefore be feasible for i to be outside the interval and the right hand side of the constraint would then be greater than or equal to $67 = ub_i$. For $X = 8$, we know that i must be one of the interval vertices. The right hand side of the constraint is then 65 as intended. For any (infeasible) configuration where $X > 8$, the constraint imposes an even stronger restriction on the upper bound of i which often cuts off the respective solutions. In particular, this example is taken from a real instance and LP solution such that the left hand side of constraint (3.37) for i was 65 while there were nine predecessors of i_a from $I(62, 65) \setminus \{i\}$. This infeasible solution is, though i is already scheduled within $[52, 65]$, cut off by the inequality since its right hand side is only 63.

3.5.6 Predecessor / Successor Set Constraints

The distance inequalities that implement the lower and upper bound constraints impose a good restriction on the number of potential predecessors and successors of an instruction. If they are strengthened using the concepts from Sect. 3.4.4, they also well restrict the sets of candidate instructions responsible to establish the lower and upper bounds. However, sometimes it may be beneficial to enforce a certain bound on the number of predecessors or successors out of a *given particular set* of instructions. In a very general fashion with particular predecessor sets $P \subseteq P^*(i)$ and successor sets $S \subseteq S^*(i)$, such upper bounding (lower bounding with '$\geq$' instead of '$\leq$') constraints can be formulated for an instruction $i \in I$ as follows:

$$\begin{aligned}\sum_{j \in P} x_{j,i} &\leq k \\ \sum_{j \in S} x_{i,j} &\leq k\end{aligned}$$

Lemma 2.4.18 in Sect. 2.4.5.2 provides an example to straightforwardly construct such inequalities.

3.5.7 Superiority Inequalities

By a *superiority inequality* we mean a very simple constraint of the form $x_{c,d} \geq x_{a,b}$ for some $a, b, c, d \in V$, $a \neq b$ and $c \neq d$, that has the interpretation $x_{c,d} = 1$ whenever $x_{a,b} = 1$, and $x_{a,b} = 0$ whenever $x_{c,d} = 0$. In general, the impact of such inequalities is rather weak. However, if one succeeds in linking rather unrelated variables due to logical implications, these constraints might help to find solutions or detect infeasibility more quickly. A practical example where this may be applied is given subsequently.

3.5.7.1 Superiority Inequalities for Overlapping Intervals

The following idea is strongly related to the one presented in Sect. 2.4.5.1. There, we were able to derive new precedences from a particular case of overlapping intervals. Here, making weaker assumptions about the relations between the involved instructions, we are at least able to derive a superiority relationship.

Theorem 3.5.10. *Let $u, v \in V$ such that $lb_u \geq ub_v - 1$. Let $i \in V$ be an instruction that is independent from u and v. Then it holds that i is a successor of v whenever i is a successor of u, and that i is a predecessor of u whenever i is a predecessor of v, i.e., $x_{v,i} \geq x_{u,i}$.*

Proof. The only case where u can precede v is if the above relation holds with equality and u attains its lower bound position while v attains its upper bound position. In all other cases, u must be a successor of v and then it is clear that, if i succeeds u, it must also succeed v and that, if i precedes v, it must also precede u.

Coming back to the first case: Suppose that i succeeds u. Then the position of i is at least $lb_u + 1 \geq ub_v$. As a consequence, i must succeed also v. Finally, suppose that i precedes v. Then the position of i is at most $ub_v - 1$. Hence, i must also precede u whose position is at least $ub_v - 1$. □

3.5.8 Variable Equality Constraints

A *variable equality constraint* is literally a constraint that enforces the equality of two linear ordering variables, i.e., $x_{c,d} = x_{a,b}$ for some $a, b, c, d \in V$, $a \neq b$ and $c \neq d$.

Being very simple, these constraints can have very different modeling semantics besides their intuitive interpretation. For example, they can be used to model an exclusive-or relation. So if an expression like $x_{a,b} + x_{c,d} = 1$ shall be formulated, then this is equivalent to enforcing $x_{a,b} = x_{d,c}$. This can be easily proven by applying the projection relation $x_{c,d} = 1 - x_{d,c}$. Again, we give a practical application.

3.5.8.1 Hall Interval-based Variable Equality Constraints

In Sect. 2.4.5.2, we already made the observation that, for any Hall interval $[a, b]$ with instruction set $I^*(a, b)$, any instruction $j \in I \setminus I^*(a, b)$ that is a predecessor (successor) of any instruction $i \in I^*(a, b)$ must be a predecessor (successor) of all the instructions in the set $I^*(a, b)$. Each such predecessor (successor) instruction j therefore has $ub_j < a$ ($lb_j > b$).

Sometimes, we may find instructions $j \in I \setminus I^*(a, b)$ being neither a predecessor nor a successor of any of the instructions in $I^*(a, b)$. It must then hold that $[a, b] \subset [lb_j, ub_j]$ since otherwise we could decide which side of $[a, b]$ is the right one for j. In this case, we can still enforce that j must be either before or after all the instructions $I^*(a, b)$ by adding the constraints:

$$x_{j,u} = x_{j,v} \qquad \text{for all } u, v \in I^*(a, b), u \neq v$$

3.5.9 NOP Difference Constraints

Like for the usual distance constraints, we are sometimes in the situation that we can derive a minimum, maximum or even exact difference between the number of NOPs before two dependent instructions i and k. For example, by carefully building the normal distance constraints using the strengthening principles presented in Sect. 3.4.4, we may find out that the distance $d_{i,k}$ can be covered by at most $b^{ub}_{i,k}$, $b^{ub}_{i,k} < d_{i,k}$, instructions and that therefore at least $N^{lb}_{i,k} = d_{i,k} - b^{ub}_{i,k}$ NOPs are necessary between the two. If all the $b^{ub}_{i,k}$ instructions are already decided to be successors of i and predecessors of k, then the usual distance constraint reduces to a minimum NOP difference constraint $n_k - n_i \geq N^{lb}_{i,k}$.

More interesting is the case where the number of instructions known to be fixed between i and k, $b^{lb}_{i,k}$, is large so that $(ub_k - lb_i - 1) - b^{lb}_{i,k} < N^{ub}_k - N^{lb}_i$. Then, the lower bound $b^{lb}_{i,k}$ allows to derive a better upper bound on the number of NOPs between i and k as is given by the NOP variable upper and lower bounds. While we cannot improve the variable bounds since we do not know whether N_i^{lb} needs to be increased or N_k^{ub} needs to be decreased, we may add the inequality $n_k - n_i \leq N_{i,k}^{ub}$ with $N_{i,k}^{ub} = (ub_k - lb_i - 1) - b_{i,k}^{lb}$.

An example where we may apply an exact NOP difference inequality of the form $n_k - n_i = N_{i,k}$ can be obtained in the same fashion as with the variable equality constraint for Hall intervals following Sect. 3.5.8.1. Since we know for any Hall interval $[a, b]$ that it is completely filled with instructions, we also know that there can be no NOP contained in $[a, b]$. Hence, the number of NOPs preceding or succeeding any pair of instructions i, j contained in $[a, b]$ must be equal. Again, we do not necessarily know how many NOPs this will be, but at least, we can make sure that the values are always equal by adding the inequality.

3.6 The Branch-and-Cut Implementation

3.6.1 Formulation as a Feasibility Problem

In Sect. 2.4, we addressed the fact that upper bounds on the issue cycles of instructions are strongly related to the global upper bound M_{ub} on the makespan. We already discussed that we may therefore consider any M in the range $[M_{lb}, M_{ub}]$ in order to obtain the corresponding issue cycle upper bounds $ub_i^M = M - d_{i,e} - 1$ that need to be respected if a schedule of length M shall be realized. A crucial observation is that new precedences $i \prec j$ can be obtained as soon as $ub_i^M \leq lb_j$ holds for some particular M. These precedences may in turn lead to additional issue cycle bound improvements and thus to further precedences. So if ub_i^M and ub_i^{M+1} are the *final* upper bounds on the issue cycle of instruction i (after propagating all transitive precedences and bound information) for assumed schedule lengths M and $M+1$ respectively, then it is possible that the difference $ub_i^{M+1} - ub_i^M$ is larger than one. Hence, conceptually fixing the current lower bound schedule length and effectively turning the optimization problem into a (series of) feasibility problem(s) can be very profitable w.r.t. search space reductions and in either finding a schedule of the current length or proving that none exists. Nevertheless, the objective function is not obsolete and can be used to steer the optimization process towards good or even optimal solutions. For instance, it can be observed that schedules being one or two cycles better than the currently best known one usually do not deviate too much from each other. Rather, there are some key instructions moved in their position such that one or more NOPs can be saved. Let σ be the best known schedule so far and let $\sigma(i)$ be the position of $i \in V$. In the solver implementation, the linear ordering variable $x_{i,j}$ is assigned the cost coefficient $c_{i,j} = \sigma(i) - \sigma(j)$ (the NOP variable coefficients are zero). The corresponding objective function (to be minimized) has the following properties:

- The coefficients are strictly negative if j succeeds i in σ. Hence, setting $x_{i,j} = 1$ and therefore placing j after i again will be rewarded by the objective function.

- The coefficients are strictly positive in the opposite case and with the same effects.
- The reward depends on how far the two instructions were placed apart before. In particular, a large distance between two instructions is considered as a stronger suggestion to keep the order of the two as before.
- No linear ordering variable will have a zero coefficient since no two instructions can have the same position in σ.

The last property is important in the following sense. While the three-dicycle inequalities suffice in order to guarantee a permutation as soon as the solution to the linear program is integral, they are - in general - quite weak when it comes to enforcing integrality. This means that, in practice, solutions to a linear program will very frequently satisfy all three-dicycle inequalities but be fractional though. Already for $n = 6$, the LP relaxation of the projected linear ordering formulation from Sect. 3.2.3 has 844 basic solutions with fractional variables (of 1560 in total) while a minimal complete description of P_{LO}^6 has only 720 vertices at all (these numbers have partially been obtained using Polymake [GJ00]). An objective function where many variables have zero or, more generally, equal cost coefficients adds symmetry to the problem and will potentially lead to more fractional variables in solutions that are optimal w.r.t. this objective function. As an illustrating example, consider the three-dicycle inequalities $x_{i,j} + x_{j,k} - x_{i,k} \geq 0$. If the objective function coefficients of the three variables are all zero, then, e.g., a binding solution $x_{i,j} = x_{j,k} = \frac{1}{3}$, $x_{i,k} = \frac{2}{3}$ is equally good as the integral binding solution $x_{i,j} = x_{i,k} = 1$, $x_{j,k} = 0$. So while we cannot guarantee that a nonsymmetric objective function will prefer integral basic feasible solutions, we can at least avoid the superfluous equivalence of fractional and integral solutions in terms of their objective function value.

3.6.2 Branching Rules

Several branching rules that were believed to be promising, especially to make 'the right' decisions in order to either find good schedules quickly or prove infeasibility of the currently assumed schedule length, were implemented. However, despite the fact that some ideas worked well on some particular instances, none of them proved to be superior on all instances. In essence, we found that a standard branching rule that selects variables with LP values close to one half can be outperformed only seldom. In the final implementation, this branching rule is applied at all subproblems with depth level less than five. After that, we first try to apply the following rules in the order of their presentation. If no variable can be found by them, we fall back to the standard rule again.

3.6.2.1 Branching on Critical NOP Variables

Sometimes it may happen that a certain instruction satisfies its lower bound position only due to a fraction of NOPs. That is, we are interested in NOP variables n_i such that $\sum_{j \in V \setminus \{i\}} x_{j,i} + n_i \geq lb_i$, but $\sum_{j \in V \setminus \{i\}} x_{j,i} + \lfloor n_i \rfloor < lb_i$. If an LP solution comprises at least one such variable n_i, we select the one that causes the largest violation of the corresponding lb_i when it is reduced to $\lfloor n_i \rfloor$. In the first created subproblem, the upper bound on the variable will be set to $\lfloor n_i \rfloor$ in the hope that it quickly proves infeasible. In the other one, the lower bound of n_i will be set to $\lceil n_i \rceil$.

3.6.2.2 Branching on Contradictory Positions

This branching rule deals with all pairs of instructions $i, k \in V, i \neq k$, such that, considering only the linear ordering variables, i precedes k, but considering also the NOP variables, i succeeds k in total. In other words, we look for linear ordering variables $x_{i,k}$ such that $\sum_{j \in V, j \neq i} x_{j,i} < \sum_{j \in V, j \neq k} x_{j,k}$, but $n_i + \sum_{j \in V, j \neq i} x_{j,i} > n_k + \sum_{j \in V, j \neq k} x_{j,k}$. Such a scenario is possible due to the big-M constraints (3.25) and (3.26). Among all variables satisfying the displayed conditions, we select the one that has its LP value closest to one half.

3.6.2.3 Branching on Equal Positions

Here, we relax the condition of the previous rule to also consider variables whose two involved instructions obtain the same position, i.e., $n_i + \sum_{j \in V, j \neq i} x_{j,i} = n_k + \sum_{j \in V, j \neq k} x_{j,k}$.

3.6.2.4 Branching on Illegal Positions

This is another relaxation. We consider all instruction pairs $i, k \in V, i \neq k$, that have a contradictory LP value assignment such that $\sum_{j \in V, j \neq i} x_{j,i} < \sum_{j \in V, j \neq k} x_{j,k}$, but $n_i \geq n_k$ (although this neither leads to equal nor contradictory positions in total).

3.6.3 Propagation at Subproblems

Whenever a linear ordering variable is set during the branch-and-bound procedure, the corresponding decision induces a new precedence in each of the two resulting subproblems. Before the first LP is solved, we propagate the transitive precedences and distance updates that result from the set branching variable. The resulting new distance constraints are added to the LP and those that have become redundant are removed. In addition, some of the preprocessing steps described in Sect. 2.4 are carried out in order to potentially further improve on the data or to detect infeasibility of the subproblem.

3.6.4 A Primal Heuristic Based on List Scheduling

In most of the cases, the solved LPs will have either fractional variables or violated three-dicycle constraints, or both, i.e., the solution is not feasible for the integer program. In any of these cases, we apply primal heuristics that construct feasible schedules by employing the current LP solution at hand in order to make decisions.

More precisely, two forward and two backward list schedules are constructed as follows. The first forward and backward list schedules obey precedences and latencies from the initial dependency DAG. The second ones obey all precedences and distances known in the current subproblem. The priority of each instruction is its distance to the artificial super sink (backward: super source) in terms of the LP solution, i.e., the corresponding number of successors (instructions and NOPs) in the forward case, and predecessors in the backward case. Further, since the precedences change with each branching step, two usual critical path list schedules (again, one forward, one backward) are carried out once at each subproblem. If a new incumbent solution is found by any of these list schedules, it is stored, the global upper bound on the makespan is updated and, if it not does not already match the currently assumed schedule length, the optimization process is restarted with an updated objective function (cf. Sect. 3.6.1). The primal heuristics are key in finding good and optimal schedules quickly. The implementation used is similar to the one discussed in Sect. 2.4.1.1.

3.6.5 Cutting Plane Separation Strategy

For the experimental evaluation, we considered two different separation strategies. The first is a minimum configuration, where the separation is mainly restricted to the three-dicycle inequalities (3CYC). Besides these, there are some constraints that are always separated as byproducts of other routines, especially of those dealing with Hall intervals and the addition of distance constraints. These are:

- Variable equality constraints w.r.t. Hall intervals following Sect. 3.5.8.1 (VEC).
- NOP difference constraints exploiting the situations mentioned in Sect. 3.5.9 (NOPD).

The second 'full' configuration activates nearly all other separation routines in order to permit an evaluation of their impact. The additionally separated inequalities are:

- Three-fence inequalities (heuristically as described in Sect. 3.2.4), if no violated three-dicycle inequalities were found (3FEN).
- Conditional bound constraints from Sect. 3.5.1 (CND).
- Conditional bound constraints exploiting transitivity implications following Sect. 3.5.2 (CNDT).
- Conditional NOP constraints as discussed in Sect. 3.5.3 (CNOP).
- Gap filling cuts as presented in Sect. 3.5.4 (GAP).
- Predecessor/successor set constraints as described in Sect. 3.5.6 and based on Lemma 2.4.18 from Sect. 2.4.5.2 (PSB).
- Superior variable inequalities based on overlapping intervals as described in Sect. 3.5.7.1 (SVC).

The capital abbreviations in parenthesis will be used in the tables of the evaluation section. The corresponding separation algorithms are, with the exception of the three-fence heuristic, all of straightforward enumerative character and of polynomial time complexity. Separation of interval filling cuts following 3.5.5 was not carried out because an exact separation routine considering all relevant intervals was found to be too time consuming. Invocations of the routine for smaller instances however indicated that these inequalities are violated frequently, like is the case for gap filling cuts as well. Still they could not be found to have a considerable impact on the hard instances discussed in Sect. 3.7.

Irrespective whether the minimum or full separation configuration was used, a branching step is enforced after at most five iterations of interleaved LP solving and separation or if no violated inequality is found.

3.6.6 Implementation with ABACUS

The solver implementation was carried out using the branch-and-cut framework `ABACUS` [Thi95]. The LP solver to be employed can be selected from a list of supported ones. For the subsequently printed results, we chose `CPLEX` in version 12.6 [CPL13]. Table 3.2 lists manually set ABACUS parameters (omitted parameters were left to their default values, or overridden by own implementations as discussed in this section).

Parameter	**value**
`NBranchingVariableCandidates`	1
`NStrongBranchingIterations`	-1 (no strong branching)
`ObjInteger`	`true`
`MaxConAdd`	$100,000$

Table 3.2: Manually set ABACUS parameters.

The choice of `ABACUS` permitted us to implement the interleaving of LP solving, the application of cutting planes, and the propagation at the subproblems of the branch-and-bound tree in a straightforward and flexible way without the need to design a complete interface to LP solvers from scratch. However, the ABACUS branch-and-bound framework is significantly slower in the creation and processing of subproblems than, e.g., explicitly developed ones (as in case of the CP solver) or even commercial ones. The reason is that ABACUS is designed to facilitate the implementation of branch-and-cut solvers (with the integration of separation routines, primal heuristics and so on) for users with only little expertise in mathematical optimization. It has a lot of functionality and flexibility at the cost of runtime performance, especially when it comes to the enumeration of a large number of subproblems.

As another matter, ABACUS does not provide a fine-tuned separation of general cutting planes for integer programs, such as, e.g. the Chvátal-Gomory cuts addressed in Sect. 1.4.6.3 which can also be found in commercial tools. Experiments with a $\{0, \frac{1}{2}\}$-cut separator developed by Alberto Caprara revealed that several cutting planes could indeed be generated from the respective linear programs solved during the course. However, the separator was itself based on the solution of integer programs and therefore too slow in order to help solving the basic block instances more quickly, while commercial IP solvers often provide fast heuristic separators. This is a particular disadvantage since it is likely that these would help in detecting infeasibility of some schedule lengths more quickly, especially because the problem to be solved w.r.t. the feasibility problems is indeed to prove that a particular polyhedron contains no integer point. Even more, some of the more complex classes of facet defining inequalities of the linear ordering polytope, such as, e.g., the Möbius ladders [GJR85a], are in fact special cases of $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts that could be recognized this way.

Despite knowing about these disadvantages, ABACUS was preferred over a commercial IP optimization software because the author did not aim at competing with the CP solver by means of black-box tools. Clearly, the sustained performance of the models derived in this chapter could possibly even be better if even more sophisticated integer programming techniques were implemented or by simply profiting from some closed-source implementation tweaks that we could not properly name. Using ABACUS, it could be made sure that the results presented in the following section are achieved by means of the models and techniques that arose from the research presented in this thesis only.

3.7 Experimental Evaluation

The approach developed in this thesis is evaluated using the same test suite that was used in the paper presenting the optimal CP approach by Malik, McInnes and van Beek [MMvB08]. Fortunately, Peter van Beek sent the instances. The set contains even roughly 17,000 instances more than were used in their experiments and that we now solved in addition. In total, the set comprises 369,861 pre- and post-register-allocation basic blocks taken from 28 application codes of the SPEC 2000 integer and floating point benchmarks. When referring to particular instances, the prefix `AR` indicates a post-register-allocation block. Solvers presented prior to the mentioned CP solver failed to solve hundreds of instances from this set. In [MMvB08], the authors report that they were able to solve all but two instances to optimality for single-issue processors within a time limit of ten minutes of CPU and system time. In our repeated experiments, that were run single-threaded on a Debian Linux system with `g++` 4.7.2 and optimization level `-O2` on an Intel Core i7-3770T processor running at 2.5 GHz and with 32 GB RAM, we found only one instance that timed out with their solver. Within the same time limit, our solver was not able to solve eleven instances using the minimum separation configuration described in Sect. 3.6.5.

Table 3.3 categorizes the instances and the computational results w.r.t. the size of the basic blocks. The third column states the number of instances that could be solved by the applied preprocessing techniques only, i.e., by proving optimality of a list schedule without solving an IP. The large numbers reflect the importance

		Branch-and-Cut				CP [MMvB08]		
Size	**#DAGs**	**Prep**	**>600s**	**>60s**	**>1s**	**>600s**	**>60s**	**>1s**
3 - 5	190,726	190,726						
6 - 10	96,807	96,803						
11 - 15	33,229	33,166						
16 - 25	23,994	23,903						1
26 - 50	15,801	15,602			2			
51 - 100	5,945	5,819	3	7	11			17
101 - 250	2,956	2,851	3	3	13			16
251 - 500	256	235	1	1	17			38
501 - 1000	105	94	2	3	33	1	2	70
1001 - 2597	42	33	2	11	33		7	41
total	369,861	369,231	11	25	109	1	9	183

Table 3.3: Size distribution of the instances, number of instances solved by preprocessing and timeouts for various time limits.

and the success of these methods for instruction scheduling while being completely independent from the final exact solution approach. In total, 74 instances more could be scheduled in less than a second compared to the CP solver. However, there are also 16 more instances that needed between one and 60 seconds and ten instances more that could not be solved within a time limit of ten minutes.

Table 3.4 lists all the instances that the solver was not able to solve within ten minutes of CPU and system time. The optimization was not always stopped exactly after this time (as is shown in the last column) because the time limit was passed to the internal timer provided by ABACUS for the IP solution phase. So if the preprocessing took a long time or multiple IPs were solved, the total time could exceed the limit given for a single IP. Column ILB gives the initial lower bound on the makespan (after the preprocessing phase). PLB denotes the lower bound that the whole solver was able to prove in the time denoted in the last column, and BEST the best solution it could find. OPT is the optimum makespan (if known) and IUB is the length of the best initially determined list schedule. We remark that the DAGs corresponding to the instances `AR-12061` and `AR-11852` are identical.

Instance `AR-4661` is the only one that could be solved by neither of the two methods. Hence, the optimum makespan is unknown. However, the list scheduler implemented into the CP solver found a solution of length 1307 and the solver could prove that no schedule with a makespan smaller than 1306 exists.

Only in a single case (`AR-3529`), the lower bound proven by the branch-and-cut solver is optimal, but an optimal solution was not found within the time limit. An optimal solution was also not found for the instances `AR-4661` and `AR-9459` but also without having proved that no better schedule can exist. In all the other cases, the optimum solution has been found, but the instances could eventually not be solved because the solver was not able to prove that no better schedule exists.

Disappointingly, this situation does not change when activating the full separation strategy as described in Sect. 3.6.5. Tables 3.5 and 3.6 show statistical data about

Basic Block	# Instr.	ILB	PLB	OPT	BEST	IUB	Time
crafty/AR-2903	495	876	878	879	879	882	632
crafty/AR-4661	713	1,301	1,303	[1,306, 1,307]	1,308	1,310	773
fma3d/6261	141	250	250	251	251	251	608
fma3d/5417	77	99	101	102	102	107	601
fma3d/6916	149	259	259	260	260	260	607
fma3d/AR-9459	860	923	925	932	938	1,039	784
jpeg/AR-3529	1,824	3,554	3,554	3,554	3,556	3,561	922
mesa/AR-11436	1,508	1,735	1,736	1,737	1,737	1,739	699
sixtrack/AR-12061	87	93	94	95	95	101	1,007
sixtrack/AR-11852	87	93	94	95	95	101	1,007
sixtrack/5960	195	195	195	198	198	210	602

Table 3.4: Instances not solved within ten minutes by the branch-and-cut solver.

the number of subproblems and LPs solved as well as the number of separated inequalities for both separation strategies and the timed-out instances. We remark again that a branching step was enforced after at most five iterations of LP solving and separation at each branch-and-bound subproblem. The reason for this is that, due to the large number of generated inequalities, the time spent at one subproblem can be sometimes very exhaustive. Further, a tailing-off strategy is not promising because the artificial objective function is not really indicative. Even more, the impact of the (transitive) implications of a branching step when re-applying some of the preprocessing techniques was found to be stronger than the impact of the several classes of inequalities in general. While it is a positive result that a large number of violated inequalities could be found by the various separation routines, these inequalities could not prove essential in determining infeasibility of the respective integer programs. On the contrary, the additional time spent for separation even led to three more timeouts for the instances `fma3d/5416`, `fma3d/6612`, and `vpr/3140`. Only for the instance `5960`, we can observe that the additional inequalities helped to prove at least the nonexistence of a schedule without NOPs more quickly. The tables also show that for some of the largest instances only a few subproblems and LPs were solved within the time limit. As discussed in Sect. 3.6.6, the speed of enumeration (as well as loading and unloading LPs etc.) is one of those weaknesses of the implementation that could easily be alleviated if a practical use of these methods was to be considered. At this point, it should also be mentioned that the preprocessing has not been extensively tuned. Especially for very large instances, an iterative application of the methods described in Sect. 2.4.4 to obtain new precedences and their transitive propagation could run for hours before reaching a fixpoint where no more precedences can be derived. The implemented solver has only very simple rules that stop this way of deriving new precedences if less than half of a percent of the precedences obtained in the very first run are obtained in the current iteration. Other preprocessing routines are not all steered and run until a fixpoint is reached. In contrast to that, the CP solver sets relative time limits for various subroutines based on the size of the instances. A more intensive tuning of parameters could therefore lead to being even more competitive.

Nonetheless, the results indicate that in proving that no schedule of a given length can exist, the relaxation of integrality ruled out to be rather a disadvantage of the IP method compared to the enumerative construction character of the CP solver. Especially for instances with a lot of symmetry, an aggressive fixing and propagation of instructions to issue cycles can detect infeasibility of all possible configurations more quickly than the solution of linear programs where it must be proven that no *fractional* solution exists that satisfies all the inequalities. Fig. 3.12 shows a typical symmetric instance with several parallel 'levels'. The instructions of one level are candidates to cover the latencies between the instructions on all other levels, and vice versa. In many cases, the symmetry breaking strategy from Sect. 2.4.4 cannot be applied because the majority of candidate pairs (or sets) of instructions have at least one predecessor or successor that violates the criterion to safely add new precedence arcs. If the lower bound obtained for an instance does not match the length of a list schedule, then several orders need to be tried in order to prove the nonexistence of a shorter schedule. Another difference to the CP solver is that the LP-solution-based branching rules and primal heuristics typically make the branch-and-cut solver more sensitive to the underlying LP solver, as LPs frequently do not have a unique optimum solution and, in this case, the objective function further depends on the reference schedule used. Different solutions, however, may lead to different branching decisions or results of the primal heuristic, potentially with impact on the solution process. Nevertheless, the presented method proved to be successful and reliable in practice for a very large range of instances. The quadratic number of linear ordering variables turned out to be no severe limitation when it comes to the solution of larger instances. The disadvantages in this sense appear to be alleviated by the opportunity to fix many of these variables, to profit from transitive precedence propagation, and to formulate tighter distance constraints having some notion of betweenness of instructions and NOPs.

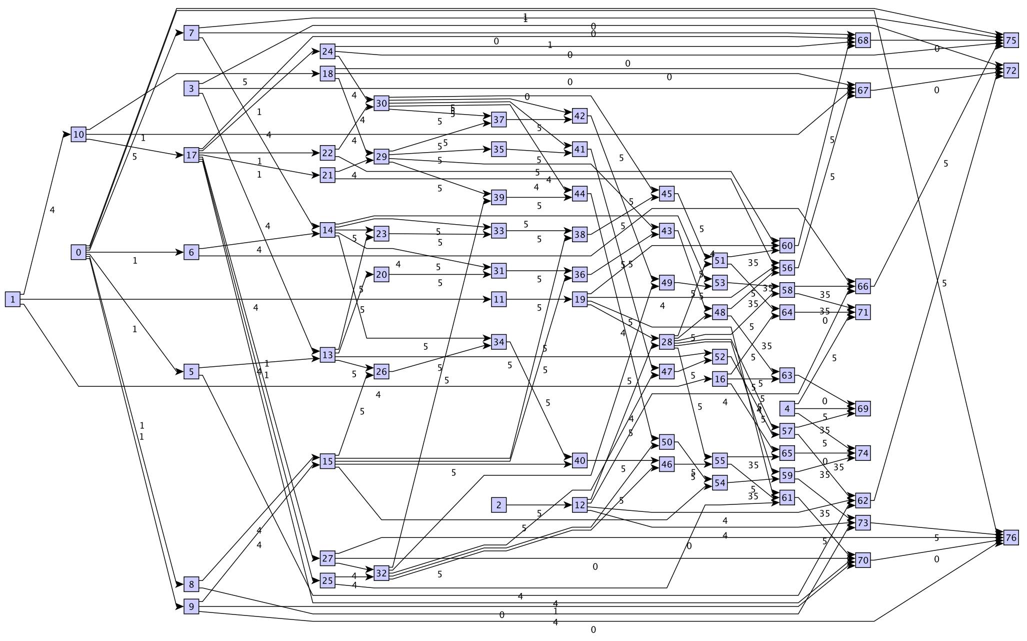


Figure 3.12: Instance `fma3d/5417` (layouted using 'yEd', a diagram editor developed and provided by the german company yWorks GmbH).

Basic Block	IPs	SUB	LPs	3CYC	3FEN	CND	CNDT	GAP	VEC	SVC	PSB	NOPD	CNOP	Time
crafty/AR-2903.txt	1	112	366	15, 726					0			142		632
crafty/AR-4661.txt	1	15	35	27, 000					0			699		773
fma3d/6261.txt	1	25, 507	60, 456	463, 563					0			0		608
fma3d/5417.txt	2	2, 205	5, 693	132, 268					6			12		601
fma3d/6916.txt	1	23, 411	53, 961	397, 058					2			0		607
fma3d/AR-9459.txt	1	8	9	2, 007					0			1, 628		784
jpeg/AR-3529.txt	1	42	85	1, 221					0			17		922
mesa/AR-11436.txt	1	17	43	3, 841					84			710		699
sixtrack/AR-12061.txt	2	22, 207	71, 521	1, 554, 133					20			25		1, 007
sixtrack/AR-11852.txt	2	22, 268	71, 723	1, 557, 721					20			25		1, 007
sixtrack/5960.txt	1	371	940	163, 691					0			0		602

Table 3.5: Solution and separation statistics for the instances not solved when using the minimum separation configuration.

Basic Block	IPs	SUB	LPs	3CYC	3FEN	CND	CNDT	GAP	VEC	SVC	PSB	NOPD	CNOP	Time
crafty/AR-2903.txt	1	78	316	21, 488	0	1, 287	6, 438	120	0	8, 728	54	135	1, 084	611
crafty/AR-4661.txt	1	6	12	9, 000	0	1, 913	23, 379	37	0	47, 698	6	431	2, 608	629
fma3d/6261.txt	1	16, 004	38, 789	290, 784	15	11, 309	15, 591	1, 349	4	13, 676	4	0	0	608
fma3d/5417.txt	2	1, 863	5, 591	124, 842	300	9, 862	30, 610	1, 024	11	43, 127	100	7	0	603
fma3d/6916.txt	1	14, 523	34, 743	251, 369	27	9, 782	12, 669	1, 991	0	9, 256	8	0	0	608
fma3d/AR-9459.txt	1	10	15	3.194	0	198	1, 068	0	0	113	0	1, 628	0	934
jpeg/AR-3529.txt	1	33	78	1.066	0	312	848	39	121	583	17	24	133	921
mesa/AR-11436.txt	1	4	4	2, 000	0	364	1, 565	29	56	3, 707	0	424	2, 972	853
sixtrack/AR-12061.txt	2	19, 536	62, 959	1, 287, 886	8, 961	97, 269	423, 934	9, 367	80	355, 462	402	16	3	985
sixtrack/AR-11852.txt	2	19, 538	62, 965	1, 287, 926	8, 961	97, 272	423, 945	9, 368	80	355, 479	402	16	3	985
sixtrack/5960.txt	2	237	816	142, 622	0	10, 831	121, 023	635	0	14, 078	36	0	0	680

Table 3.6: Solution and separation statistics for the instances not solved when using the full separation configuration.

Chapter 4

Offset Assignment

This chapter introduces and motivates the offset assignment problem which is the second compiler optimization problem intensively dealt with in this thesis. The offset assignment problem consists of two particular subproblems whose connection is discussed in detail. The set of feasible solutions to these subproblems are characterized by different models and some new transformations of and extensions to these models are presented. These characterizations build the basis for the novel exact integer programming approaches to offset assignment that are presented in the subsequent chapter.

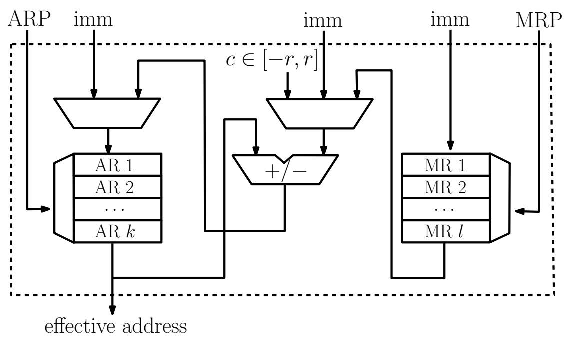
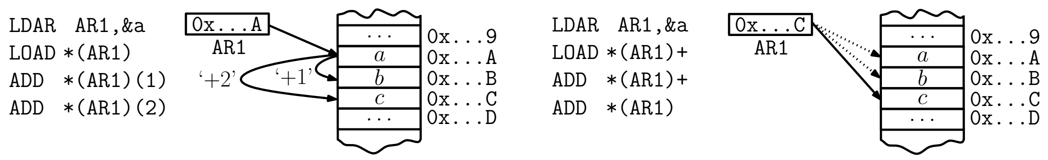


Figure 4.1: Schematic depiction of an address generation unit (AGU). It is a close adaptation of the one shown in [Leu03].

4.1 Motivation

The offset assignment problem has its origin in the field of address code generation for digital signal processors (DSPs) and was first discussed by Bartley [Bar92] in 1992. In contrast to general purpose processors, the DSPs at that time, such as e.g. the Motorola DSP56k or the TI TMS320C2x series, often did not provide so-called *base-plus-offset* addressing modes. Such addressing modes build the effective address of a memory operand by adding an immediate offset to a base address that is stored in a register. By using varying offsets, several spatially close memory locations can be referenced without the need to issue additional instructions that change the base address register. However, in this case, the offset has to be explicitly passed as an instruction operand, i.e., bits need to be reserved for it. Supporting large offsets therefore results in long instruction widths. This is in conflict with specialized processor designs that often aim at saving silicon area and, in general, costs. This causal relationship makes the task to generate optimal address code for processors with limited indirect addressing capabilities an interesting challenge for today's application-specific processor designs as well.

While base-plus-offset addressing is often not supported, DSPs and other specialized Harvard architectures usually provide an address generation unit (AGU) supporting pointer arithmetic to be done in parallel to the main data path. If exploited properly, the additional hardware, that is schematically depicted in Fig. 4.1, can help to at least partially compensate for the drawbacks of absent base-plus-offset modes. It supports instructions that permit to manipulate an address register (AR) in the same clock cycle as another instruction referencing it. Either, the modifications are encoded implicitly (effectively moving the encoding of the offset into the instruction opcode) or the instructions permit to add (subtract) values within a small architecture-dependent *auto-modify range* $[-r, r]$ to (from) the address held in the AR [Leu03]. In the special case $r = 1$, we will speak about *autoincrement* and *autodecrement* instructions. AR modifications by absolute values larger than r however still need additional explicit address arithmetic. As a further feature, a processor may provide a modify register (MR) file that can be used to add or subtract values to or from the addresses stored in ARs in parallel to another instruction.



```
LDAR  AR1,&a
LOAD *(AR1)
ADD  *(AR1)(1)
ADD  *(AR1)(2)
```

```
LDAR  AR1,&a
LOAD *(AR1)+
ADD  *(AR1)+
ADD  *(AR1)
```

Figure 4.2: Indirect addressing in base-plus-offset mode (left) and with autoincrement instructions (right).

However, any change to the value stored in a MR also needs an additional load or arithmetic instruction.

Let us consider a simple example where three values a, b and c shall be summed up and the memory layout is like the one depicted in Fig. 4.2. Consider a usual `ADD` instruction that adds the value stored at a specified memory address to the accumulator. If base-plus-offset addressing was supported, we can load, e.g., the address of a into an AR and then reference b and c using offsets one and two, respectively. This is depicted in the left of Fig. 4.2. Without base-plus-offset addressing (right image), an AR must always point exactly to the address that shall be referenced. Starting again at the address of a, we can increment the address immediately after loading a into the accumulator using an autoincrement version of the instruction (denoted with suffix `+`). The same can then be done for b, so that the full computation can also be done without additional instructions that ever explicitly manipulate the AR. However, in general, this only works as long as the distances between two variables that are subsequently accessed by the same AR are within the modify-range r.

With the mentioned specialized instructions at hand, one can consider the complexity of indirect addressing to have been moved from hardware to software, relying on compilers to exploit the processor's capabilities for fast memory addressing. In statically allocated memory, such as a local function's stack, the storage order of the program variables may be freely chosen by the compiler. Further, an address register responsible for each of the accesses needs to be determined. The first task is literally the 'real' offset assignment and the second one is called *address register assignment* (ARA). Clearly, these two tasks are interdependent, because the stack memory layout determines whether a particular access transition of an AR is within auto-modify range r or not and this determines the quality of a particular ARA. Optimal exploitation of the processors auto-modify capabilities asks for the optimal solution of both interdependent problems. Conversely, address computation overhead may result from two main issues. An inappropriate storage layout may necessitate additional explicit address arithmetic instructions for 'jumps' to addresses that have a distance larger than r. Moreover, if the processor provides multiple ARs, a poor choice of the ARs responsible for particular accesses may result in superfluous immediate AR loads and also unnecessary 'jumps'. Since address calculations make up a significant part of machine instructions and often need to be done repeatedly within loop structures [LBM98], optimizing these decisions may considerably reduce the code size and speed up the program at the same time. Indeed, various experimental studies [HABT11, JM13, Leu03, ML14] show that optimized configurations lead to significant savings in practice.

During compilation, the instruction scheduling phase determines the *access sequence* to program variables (operands). It can be extracted by simply concatenating the referenced variables of each three-address-code instruction $c = a \ op \ b$ in the order $a \ b \ c$. For example, the program fragment in the left of Fig. 4.3 refers to the variables $\mathcal{V} = \{a, b, c, d, e, f, g\}$ that are accessed in the order $S = a \ b \ c \ g \ c \ f \ c \ e \ c \ c \ f \ d$. Tab. 4.1 shows pseudo machine codes for the code fragment and three potential stack layouts A, B, and C (now depicted horizontally) of $\mathcal{V}$. Layout A complies to the order of first use of the variables in S. On a processor with only a single AR, it would require six explicit address arithmetic instructions (`ADAR` and `SBAR`). An optimized layout (B) already reduces the necessary number of such instructions to three by increasing the use of autoin-/decrement instructions. If the memory layout is optimized for a use of two ARs (C) and also an optimal AR assignment is computed, it becomes possible to cover the access sequence even without any explicit address arithmetic at all. Assuming unit costs for an immediate AR load and for an address arithmetic instruction, the optimal total cost with one AR is four, while, with two ARs, it is two. Notably, layout A and B have no register assignment that leads to a total cost smaller than three with two or more ARs.

```
c = a + b;
f = g - c;
c = c - e;
d = c * f;
```

Figure 4.3: A sample code fragment.

Instruction	AR1
LDAR AR1, &a	&a
LOAD *(AR1)+	&b
ADD *(AR1)+	&c
STOR *(AR1)+	&g
LOAD *(AR1)-	&c
SUB *(AR1)	
ADAR AR1,2	&f
STOR *(AR1)	
SBAR AR1,2	&c
LOAD *(AR1)	
ADAR AR1,3	&e
SUB *(AR1)	
SBAR AR1,3	&c
STOR *(AR1)	
ADAR AR1,2	&f
MUL *(AR1)	
ADAR AR1,2	&d
STOR *(AR1)	

$A = a \ b \ c \ g \ f \ e \ d$

Instruction	AR1
LDAR AR1, &a	&a
LOAD *(AR1)+	&b
ADD *(AR1)	
ADAR AR1,2	&c
STOR *(AR1)-	&g
LOAD *(AR1)+	&c
SUB *(AR1)+	&f
STOR *(AR1)-	&c
LOAD *(AR1)	
ADAR AR1,3	&e
SUB *(AR1)	
SBAR AR1,3	&c
STOR *(AR1)+	&f
MUL *(AR1)+	&d
STOR *(AR1)	

$B = a \ b \ g \ c \ f \ d \ e$

Instruction	AR1	AR2
LDAR AR1, &a	&a	
LOAD *(AR1)+	&b	
ADD *(AR1)+	&c	
STOR *(AR1)		
LDAR AR2, &g		&g
LOAD *(AR2)-		&e
SUB *(AR1)+	&f	
STOR *(AR1)-	&c	
LOAD *(AR1)		
SUB *(AR2)		
STOR *(AR1)+	&f	
MUL *(AR1)+	&d	
STOR *(AR1)		

$C = a \ b \ c \ f \ d \ e \ g$

Table 4.1: Pseudo machine codes for the code fragment from Fig. 4.3 assuming different memory layouts A, B and C and either one (A and B) or two (C) available address registers.

4.2 Problem Definitions and Related Work

The *General Offset Assignment* (GOA) problem is defined for processors with k address registers, l modify registers and an auto-modify range of r. Given an access sequence $S = \{s_1, s_2, \ldots, s_m\}$ of program variables $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, it asks for a stack memory layout of the variables, i.e., a permutation $\pi : \mathcal{V} \to \{1, \ldots, n\}$, and an assignment $\gamma : S \to \{1, \ldots, k\}$ of accesses to address registers exploiting the auto-modify range r such that the number of accesses requiring extra address arithmetic instructions is minimum. If modify registers are considered, it must further be decided for each MR, at which points in time it is assigned a new value and when this value is added to (or subtracted from) one of the address registers. MRs lead to a larger solution space and to a higher optimization potential but also considerably increase the modeling complexity. They have been seldom integrated into practical approaches for address code generation. For $k = 1$, Wess and Gotschlich [WG97] preload MRs with static values $\{-2, -1, 1, 2\}$ that cannot be changed during program execution. This is conceptually equivalent to an auto-modify range r of two. Leupers and David [LD98] provide a concept to incorporate MR optimization into the fitness function of their genetic GOA algorithm that has however some other restrictions as will be discussed below.

Most of the literature considers special cases of the problem, where either $r = 1$, $k = 1$ or both. For $k = 1$, it is called the *Simple Offset Assignment* (SOA) problem. It reduces to the task to find a stack memory layout that allows as many accesses as possible to be performed by auto-modify instructions on the single available address register. It was first considered by Bartley [Bar92] in 1992. He proposed to model the problem using an *access graph* that we will discuss in Sect. 4.3. Using this representation, Bartley recognized a close relationship of SOA (with $r = 1$) to the Maximum Weight Hamiltonian Path problem and developed a first greedy heuristic to solve SOA in this fashion. In subsequent research, Liao [Lia96] showed that SOA is equivalent to a Maximum Weight Path Cover problem and gave a formal proof of its strong $\mathcal{NP}$-hardness. Consequently, the more general problem variants are equally $\mathcal{NP}$-hard. Based on these results, Liao proposed a simpler and faster heuristic producing solutions with the same quality as Bartley's and also a first exact branch-and-bound procedure. In 1996, Leupers and Marwedel [LM96] proposed to use a tie-break function for access graph edges with equal weights within Liao's heuristic. One year later, Leupers and David presented the already mentioned genetic algorithm for GOA [LD98] that can also be used for $k = 1$. Atri et al. [ARK01] developed an incremental algorithm that tries to successively improve a known feasible solution. These already mentioned algorithms were subject to an exhaustive experimental comparison by Leupers [Leu03] in 2003. It revealed only small differences in the quality of their solutions. However, the performance of the heuristics relative to the optimum solutions could only be verified for some small instances using Liao's branch-and-bound procedure. The corresponding benchmark set, called *OffsetStone*, is since then the standard reference for performance measures. In 2008, another tie-break-heuristic for Liao's algorithm has been proposed by Ali et al. [AEBS08] that was later [SEB12] also evaluated with OffsetStone. One of the exact methods for SOA presented in this thesis, and prior published in [JM13], was then the first approach capable to solve larger instances to optimality, so that the quality of heuristics could be broadly evaluated for the first time.

The general problem variant for multiple ARs has also been studied mostly for the case $r = 1$. In principle, the two interdependent subproblems may be solved in any order. Many of the proposed algorithms first create an ARA by partitioning the set of program variables w.r.t. the available ARs and solve then the SOA problem for each of them. This has the advantage that available SOA algorithms can be reused in order to solve GOA. So while SOA seems to be an oversimplified problem at the first glance, it reflects a real world problem. However, the described strategy inherently constrains all accesses to a particular variable to be performed by the same AR. This may preclude optimal results as is extensively discussed by Huynh et al. [HABT11]. In the same article, the authors evaluate different combinations of ARA and SOA algorithms with the approach by Sugino et al. [SINF96] performs best in their experiments. Sugino et al. partition the variables iteratively by applying, in each iteration, a minimum-cut heuristic that repeatedly invokes a SOA algorithm to estimate the quality of the partitioning. As opposed to Bartley's and Liao's greedy constructive methods, their SOA algorithm removes edges from the access graph until a variable ordering can be trivially derived. The overall procedure is computationally intensive but the additional effort appears to not pay off in terms of quality compared to other heuristics on OffsetStone [ML14]. Besides his algorithm for SOA, Liao [Lia96] also proposed a heuristic for GOA that however needed some manual parameter specification. In 1996, Leupers and Marwedel [LM96] proposed a different heuristic that outperforms Liao's on their random test instances. Their method was also used to generate initial populations for the genetic GOA algorithm from [LD98] that, however, also assigns all accesses to a particular variable to the same AR. The results in [ML14] suggest not to partition the variables a priori, but to first compute a memory layout for them and an address register assignment afterwards. The article also contains a correction of the only previous exact approach by Ozturk et al. [OKT06] that was originally designed for arbitrary auto-modify ranges r and that can also be extended to deal with MRs. However, as the experiments in [ML14] show, the (fixed) method is not capable to solve larger instances, even for $r = 1$. Also for the original version, the authors reported running times for moderate instance sizes that do not suggest to use their method in a production compiler. It suffers from a quickly growing number of variables and constraints and does neither exploit the combinatorial structure nor symmetries inherent to the problem.

In this thesis, alternative GOA formulations for $r = 1$ and also for arbitrary r are presented in Sect. 5.2. Similar to the exact SOA methods that are discussed in Sect. 5.1, the methods for $r = 1$ can solve nearly all of OffsetStone's instances to optimality within short time frames and were used to evaluate the quality of GOA heuristics in [ML14]. The method for $r \geq 1$ takes advantage of the $r = 1$ approach and still remains relatively moderate in size. It is not as successful, but made it possible to perform the first experiments for $r > 1$ on a larger set of instances and to study the effect of exploiting larger auto-modify capabilities on the total address computation costs [Mal14]. Further, a commonly observed criticism associated with GOA is addressed, namely that it is not clear how it relates to operand reordering techniques such as, e.g, [RP99, ARK00, CK02], which may also result in reduced address computation overhead. In Sect. 4.4.3, a method to integrate commutativity-based operand reordering into the address register assignment part of the optimization process is presented. It has also been published in [Mal14].

We close this section by mentioning some specialized and integrated variants of the problem that were subject to research but not directly comparable to the methods developed in this thesis. Lorenz et al. [LKB$^+$01] consider offset assignment in the context of specialized DSPs with wide memory and where accesses do not refer to single memory words but to all variables of a previously specified group simultaneously. Eriksson [Eri11] proposed a dynamic programming algorithm to integrate scheduling, AR and offset assignment. However, the algorithm could solve only small instances. A different idea named *variable coalescing* is to share storage locations among variables whose lifetimes do not overlap. Ottoni et al. [OOAL06] presented a first heuristic which was followed by another one and an ILP formulation by Salamy and Ramanujam [SR07, SR12]. However, for the exact approach, the instances solved did again not exceed sizes of about 30 variables due to the running time of the solver.

4.3 Modeling of (Optimal) Memory Layouts

One part of the problem is to derive a stack memory layout of the variables, the literal 'offset assignment'. In case of the Simple Offset Assignment problem, it is even the only task to do. This is true, since for SOA there is only one AR that must be used to perform all the accesses. Hence, its series of accesses is predetermined (it is simply equal to the access sequence) and the cost of each access transition only depends on the locations of the variables.

For the case $k = 1$ and $r = 1$, Bartley [Bar92] proposed to model the problem using an *access graph* $G = (V, E)$. The set of vertices V corresponds to the variables $\mathcal{V}$ and there is an edge $e = \{u, v\} \in E$ with weight $w(e) = c$ if the variables u and v appear subsequently in the access sequence for $c > 0$ times. Fig. 4.4 shows an example for the access graph belonging to the access sequence of the code fragment from Fig. 4.3.

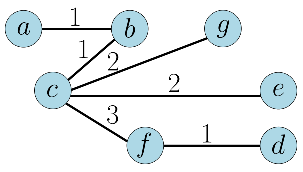


Figure 4.4: An access graph corresponding to the access sequence $S = a\ b\ c\ g\ c\ f\ c\ e\ c\ c\ f\ d$ of the code fragment from Fig. 4.3.

4.3.1 Hamiltonian Path and Path Cover Characterizations

For $r = 1$, an access transition from variable u to variable v can be performed by an AR using autoin-/decrement instructions if and only if u and v are adjacent in the memory layout. It is a natural idea to model such adjacencies also by adjacencies in the access graph, i.e., u and v shall be neighbors in the memory layout if the edge $\{u, v\} \in E$ is selected. There is an 'all-or-nothing' condition in that, if an edge is selected, all the corresponding access transitions in the access sequence (whose number is equal to the edge weight) can be performed without the need for additional instructions. The objective is therefore to maximize the total weight of selected edges subject to the constraint that they can be combined to form a memory layout (a linear sequence) of the program variables.

As already stated, Bartley [Bar92] characterized the memory layout as a Hamiltonian path and Liao [Lia96] as a path cover. A Hamiltonian path P is a simple path visiting each of the vertices V exactly once. Hence, all but two vertices of P have exactly two neighbors and the two exceptional end points of P have only one neighbor. P consists of exactly $|V|-1$ edges from E. Clearly, it is easy to interpret a Hamiltonian path as a memory layout by simply arranging the variables $\mathcal{V}$ according to the neighborships of their corresponding vertices in the path P. There is some symmetry however, because it is possible to traverse P in two directions, i.e., one Hamiltonian path has two associated memory layouts of the same quality. A path cover C is a cycle-free collection of edges $e \in E$ such that each vertex $v \in V$ has *at most* two neighbors in C. This definition permits isolated vertices, i.e., vertices do not necessarily need to be at all 'visited' or 'covered' by one of the edges. Fig. 4.5 depicts an example for both characterizations using the access graph from Fig. 4.4. We might interpret isolated vertices as 'paths of length zero', in which case we can say that a path cover is a collection of paths such that each vertex $v \in V$ is covered exactly once. There is even more symmetry compared to the Hamiltonian path characterization, since each possible concatenation of the paths of a path cover C can be mapped to a memory layout of the same quality.

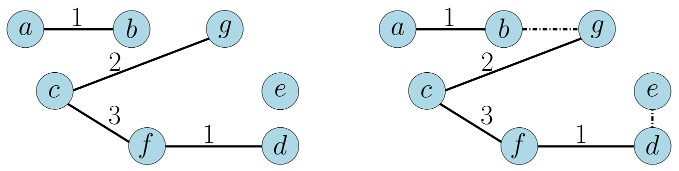


Figure 4.5: A path cover (left) of the access graph from Fig. 4.4 and a concatenation of its paths (right, dash-dotted dark additional zero-weight edges) to a Hamiltonian path that corresponds to memory layout $B = a\ b\ g\ c\ f\ d\ e$ from Sect. 4.1.

The central issue with the Hamiltonian path characterization is that, if only the edges with weight at least one are present, it is not guaranteed that the graph admits a Hamiltonian path. This is why Bartley proposed to make the graph complete by adding zero-weight edges for all the pairs of variables that are never adjacent in the access sequence. Adding zero-weight edges does not change the set of solutions and it does also not alter their weighting. Each zero-weight edge contained in a Hamiltonian path exactly corresponds to the concatenation of two paths of a path cover. However, since the selection of zero-weight edges cannot improve a solution (they deliver no additional access transitions that can be done with autoin-/decrements), Liao concluded that it must be possible to encode optimum solution by considering the nonzero-weight edges only which led to the path cover characterization.

We summarize these observations in the following formal statements.

Lemma 4.3.1. *Let $G = (V, E)$ be an access graph and let $\mathcal{P}$ be a maximum-weight path cover of G. If $|\mathcal{P}| > 1$, then for any two end-vertices p, q of different paths $P, Q \in \mathcal{P}$, the number of access transitions between p and q is zero.*

Theorem 4.3.2. *Let $G = (V, E)$ be an undirected graph and let $G' = (V, E')$ be the complete graph that results by adding a zero-weight edge for every edge that is not in G. Then there exists a maximum-weight path cover P of weight $w(P)$ in G if and only if there exists a maximum-weight Hamiltonian path P' of weight $w(P') = w(P)$ in G'.*

Proof. Let P be a maximum-weight path cover in G. Clearly, if P consists only of a single path, then P is also a maximum-weight Hamiltonian path. So let P consist of $k > 1$ disjoint paths. By construction, no two paths of P share an end vertex, since otherwise they would build a larger path. Hence, by Lemma 4.3.1, there exists an Hamiltonian path P' in G' that consists of P and $k - 1$ additional edges with zero weight. By construction, P' has the same weight as P. Suppose now that P' is not a maximum-weight Hamiltonian path in G', i.e., there exists a different path Q with weight $w(Q) > w(P')$. However, then Q, without its zero-weight edges, is also a maximum-weight path cover in G with weight greater than $w(P)$. This is a contradiction to the assumption that P is maximum. Conversely, a maximum-weight Hamiltonian path P' in G' yields (by removing zero-weight edges) directly a path cover P of the same weight, and there cannot be a better one, because this would contradict the optimality of P'. $\square$

Both modeling approaches have their advantages and disadvantages. Typically, variables are involved in rather few and local computations with some particular (but, in most of the cases, not all) other variables. Hence, it is natural to assume that access graphs are rather sparse. Algorithms will usually have to consider each vertex and each edge of the access graph at least once such that their running time directly depends on the cardinality of the respective sets. Making the graph complete results in a number of edges that is quadratic in the size of the variable set, even though the number of nonzero-weight edges could be only linear. This may unnecessarily slow down algorithms [Lia96]. However, as already indicated, the path cover characterization also adds more symmetry to the set of feasible solutions. This is an issue especially for exact solution approaches. There is also a significant difference when formulating integer programs for the two models which will be discussed in Sect. 5.1.

4.3.2 Explicit Assignment of Positions

If the processor supports offset ranges r strictly larger than one, simple vertex adjacencies are not anymore sufficient in order to decide on the costs of access transitions. In particular, it is misleading to adjust the edge weights $w(\{u, v\})$ of the access graph to the number of occurrences of u and v with distance at most r in the access sequence S. Rather, the auto-modify instructions can be applied as soon as there is a *path* from u to v of length smaller or equal to r in a memory layout. To assign weights to paths instead of edges and to then select paths based on these weights is however not a promising strategy, especially since the possible number of paths to consider may be exponential in the number of variables.

These conditions make the modeling of the associated optimization problem and the derivation of integer programming formulations much more complicated. It appears that it is necessary to explicitly encode the position of each of the program variables and then to construct and evaluate solutions based on the absolute distances between them. One way to do this is the famous *assignment problem* [BDM12]. A very elegant model for SOA and arbitrary r that is based on the quadratic assignment problem has been proposed by Wess and Gotschlich [WG97], although they did not try to solve it exactly. Their method exploits that one useful property persists due to the $k = 1$ restriction: For any pair of variables $u, v \in V$, either all or none of the access transitions (that is still predetermined by the access sequence) can be done using auto-modify instructions. This preserves the ability to work with the (adjacency matrix associated to) the access graph and with static cost coefficients. However, it makes products of variables (and therefore the quadratic objective function) indispensable. The formulation has then $\Omega(|V|^4)$ variables that become a limitation for larger instances. The model by Ozturk et al. [OKT06] has an assignment-based character as well. However, as discussed in [ML14], their formulation was flawed and is also not applicable to a wider range of instances. In Sect. 5.2, a different (though still assignment-based) and completely linear formulation for GOA will be presented that can then also be used for the $k = 1$ case.

4.4 Optimal Address Register Assignment

The address register assignment problem becomes relevant as soon as the number k of address registers is larger than one. In this case, the series of accesses performed by an AR is no longer predetermined by the access sequence. In principle, each AR may be used for each access and it may be beneficial to use even different ARs for two accesses to the same variable [HABT11]. As a consequence, two subsequent accesses performed by the same AR need not be subsequent in the access sequence anymore. Conversely, two subsequent accesses in the sequence do not necessarily appear as a real access transition of one of the ARs. So while it remains true that either all or none of the access transitions between two variables u and v *can* be done by auto-modify instructions (since this is just a matter of their distance in the memory layout), this property is of no more use. This degrades the access graph to a more or less useless data structure.

In case that a memory layout of the variables is already *given*, an optimal assignment of accesses to registers can be computed in polynomial time. This was shown by Gebotys [Geb97, Geb99] who provided a minimum-cost circulation algorithm that we will now discuss in detail. After that, newly developed transformations and extensions of this model that have been published also in [ML14] and [Mal14] are presented.

4.4.1 Gebotys' Circulation Technique

Suppose for now that a memory layout $\mathcal{L}$ of the program variables $\mathcal{V}$ has already been fixed and we are now asked to compute an optimal address register assignment for k address registers w.r.t. $\mathcal{L}$ and the input access sequence S.

Gebotys' circulation network contains a vertex for each access in S and a directed arc for each pair of accesses u, v such that v succeeds u in S. Let V_S be an ordered set of vertices associated with the accesses in sequence S and consider additional artificial 'source' (s) and 'sink' (t) vertices. Then the vertex set V_N of the network $N = (V_N, A)$ is given by $V_N = V_S \cup \{s, t\}$ and the flow arc set A is composed as the union of the arc sets $\{(s, v) \mid v \in V_S\}$, $\{(v, w) \mid v, w \in V_S,\ v < w\}$, $\{(v, t) \mid v \in V_S\}$, and the circulation arc (t, s).

As a small example, let $\mathcal{V} = \{a, b, c, d\}$, $S = a\ d\ c\ c\ a\ b$, and assume $\mathcal{L} = d$ - a - c - b (which is optimal for $k \geq 2$ ARs). Fig. 4.6 shows the network associated with this instance.

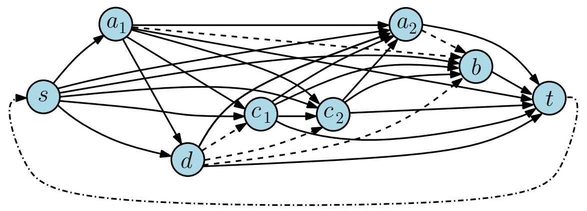


Figure 4.6: Minimum cost circulation network for $S = a\ d\ c\ c\ a\ b$ assuming $\mathcal{L} = d$ - a - c - b.

The cost of an arc between two accesses is zero if and only if the two associated variables are equal, or adjacent in $\mathcal{L}$ (solidly drawn in Fig. 4.6). Otherwise the cost c_A of an address arithmetic instruction is associated with the arc (drawn dashed). All costs of arcs leaving s or entering t are also zero and the dash-dotted drawn arc (t, s) has the cost c_L of an immediate address register load.

Each vertex is constrained to receive and supply one unit of flow and the capacity of all arcs is one, except for the arc (t, s) that has capacity k. Hence, the maximum possible flow in this circulation network is k units and each unit of flow leaving s essentially delivers a path of accesses before it proceeds to t. If the selection of these paths is based on a minimum-cost criterion then each of the resulting paths can be interpreted as an optimal series of accesses performed by an AR.

Minimum-cost circulations can be established in polynomial time using either combinatorial algorithms (e.g. [Tar85]) or linear programming. Optimal LP solutions will always be integral due to the unimodularity property of the constraint matrix associated to minimum-cost circulation problems [Law76].

Gebotys also gave an LP formulation of her approach. Let $y_{u,v}$ be a flow variable for each arc $(u, v) \in A$ and $c_{u,v}$ its associated cost. Gebotys' LP formulation is then:

$$\begin{array}{llll}
\min & \sum\limits_{(u,v)\in A} c_{u,v} y_{u,v} & & \\
\text{s.t.} & \sum\limits_{(v,w)\in A} y_{v,w} - \sum\limits_{(u,v)\in A} y_{u,v} & = 0 & \text{for all } v \in V_N \qquad (4.1)\\
& \sum\limits_{(v,w)\in A} y_{v,w} & = 1 & \text{for all } v \in V_S \qquad (4.2)\\
& \sum\limits_{(u,v)\in A} y_{u,v} & = 1 & \text{for all } v \in V_S \qquad (4.3)\\
& y_{u,v} & \geq 0 & \text{for all } (u,v) \in A, (u,v) \neq (t,s)\\
& y_{u,v} & \leq 1 & \text{for all } (u,v) \in A, (u,v) \neq (t,s)\\
& y_{t,s} & \geq 1 & \\
& y_{t,s} & \leq k &
\end{array}$$

The restriction of the in- and out-degrees of all vertices $v \in V_S$ to one by constraints (4.2) and (4.3) highlights the aforementioned 'path selection' character of this model. Actually, they make the preceding flow conservation constraints (4.1) obsolete for all vertices except t since any unit of flow sent from s to satisfy the equations must finally arrive at t and will then be sent back using the circulation arc (t, s). Lower bounds on the flow on out- (in-) arcs of the source (sink) are not necessary since the former (latter) must be satisfied due to the in- (out-) degree equation of the first (last) access vertex.

When using combinatorial algorithms, the restrictions that the incoming and outgoing flows of each vertex $v \in V_S$ are exactly one need to be enforced using artificial gadgets like depicted in Fig. 4.7. Each vertex $v \in V_S$ is modeled by a vertex v_i for the incoming and a vertex v_o for the outgoing flow. They are connected by an arc that has lower bound and capacity one, and cost zero. This leads to a doubling of the number of vertices and adds $|V_S|$ additional arcs.

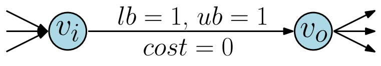


Figure 4.7: Gadget to realize a lower and upper bound of one on the incoming and outgoing flow of a vertex $v \in V_S$.

4.4.2 An Equivalent Minimum-Cost Flow Model

Further inspection of the problem and basic results from network flow theory allow for a simple transformation of Gebotys' model into a usual min-cost flow problem where the circulation arc (t, s) is removed. Its cost (the cost of an immediate AR load) can be instead installed on every s-leaving arc (that are now drawn dash-dotted). This yields the same result that the cost is paid as soon as an additional register is used to cover the access sequence. The restriction to not use more than k registers (the former capacity of the arc (t, s)) can be applied over the total flow on all s-leaving arcs instead. In this manner, the vertices s and t will be a real source and sink, respectively, and the lower bounds and capacities on all flow arcs are zero and one. Fig. 4.8 shows the transformed network corresponding to the example from Fig. 4.6, and Fig. 4.9 depicts an optimal solution (assuming $c_L = c_A$).

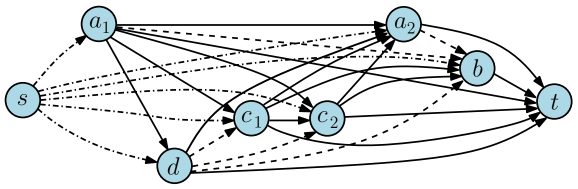


Figure 4.8: Minimum cost flow network for $S = a\ d\ c\ c\ a\ b$ assuming $\mathcal{L} = d$ - a - c - b.

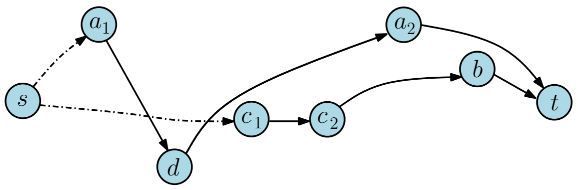


Figure 4.9: An optimal solution to the example from Fig. 4.8.

After the transformation, still combinatorial algorithms as well as linear programming can be used in order to obtain integral solutions in polynomial time. When using a combinatorial algorithm, an additional gadget similar to the one from Fig. 4.7 in Sect. 4.4.1 (with capacity k and no incoming arcs to v_i) must be installed to realize the upper bound on the total flow emanating from the source. The LP formulation corresponding to the described min-cost flow problem is:

$$\begin{aligned}
\min \quad & \sum_{(u,v)\in A} c_{u,v} y_{u,v} & & \\
\text{s.t.} \quad & \sum_{(v,w)\in A} y_{v,w} & = 1 \qquad & \text{for all } v \in V_S \\
& \sum_{(u,v)\in A} y_{u,v} & = 1 \qquad & \text{for all } v \in V_S \\
& \sum_{v\in V_S} y_{s,v} & \leq k \qquad & \\
& y_{u,v} & \geq 0 \qquad & \text{for all } (u,v) \in A \\
& y_{u,v} & \leq 1 \qquad & \text{for all } (u,v) \in A
\end{aligned}$$

4.4.3 Taking Commutativity Into Account

Sometimes, address computation overhead can also be avoided by simply reordering the accesses to operands where this is possible due to the commutativity of the respective operations. For example, assume that variable b was last accessed and `c = a + b` is the next instruction. Then it is typically beneficial to access b first if a and b are not adjacent in the stack layout.

Reordering opportunities can be incorporated into the general approach by Gebotys and also into the just described flow model. If u and v are the two operands belonging to a commutative instruction, one can replace the corresponding flow arc $(u, v) \in A$ by a flow *edge* $\{u, v\}$ that can be used in both directions (or, equivalently, add a reversed flow arc (v, u)). Since we may reasonably assume to have three-address-code instructions only, the model guarantees that each vertex of the network has at most one neighbor that is adjacent by an edge rather than by an arc. At these particular vertex pairs, the flow in the network is then permitted to also move 'backwards' while the constraints that each vertex must have exactly one incoming and outgoing unit of flow will preserve the overall feasibility of the model and the correctness of its solutions.

Suppose the access sequence $S = a\ d\ c\ c\ a\ b$ from the previous subsection is stemming from the computations `c = a * d; b = c * a`. Then both operations are commutative and the arcs (a_1, d) and (c_2, a_2) in the flow network become edges in the proposed methodology. Indeed, sending flow backwards along these edges allows for a better solution with one a single AR in use and total cost only one as is depicted in Fig. 4.10.

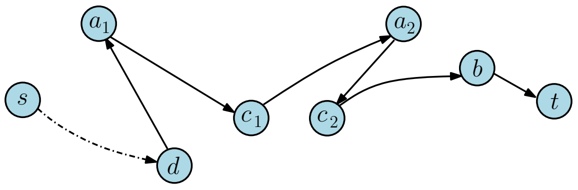


Figure 4.10: An optimal solution to the example from Fig. 4.8 exploiting commutativity.

4.4.4 Useful Properties of the Model

Gebotys' technique has some advantages that should be highlighted. First of all, it does not name registers. This means that it captures the symmetry that, once an optimal set of access transition paths is determined, each path could be realized by each AR. As long as the computed paths are assigned to different ARs, it does not matter which AR is the concrete one to perform the respective series of accesses. This property also results in the fact, that the numbers of variables, $(|S|+2)\cdot(|S|+1)/2$, and nontrivial constraints, $2|S|+1$, of the equivalent flow model only depend on the length of the access sequence S, but are completely independent from the number of registers k.

Chapter 5

Novel Integer Programming Approaches to Offset Assignment

This chapter presents novel exact integer programming approaches to several variants of the offset assignment problem that are built on the basic characterizations discussed in the previous chapter. The various models discussed led to a number of solver implementations that are presented and evaluated on a large benchmark set. The implementations are also employed to derive first results on the effects of particular processor capabilities w.r.t. the offset assignment costs.

5.1 Simple Offset Assignment

In this section, we only consider the case where $r = 1$, i.e., only autoin-/decrement instructions are available. For $r > 1$, we will however develop a formulation for the General Offset Assignment problem in Sect. 5.2, that can then also be used for the case of only a single address register.

As discussed in Sect. 4.3.1, we design integer programming formulations in terms of either path covers of an access graph, or Hamiltonian paths of completed access graphs. The main commonality in the definition of both, path covers and Hamiltonian paths, is their cycle-freedom. This is also reflected in the corresponding integer programming formulations and the main issue when it comes to their practical performance.

5.1.1 Path Cover Formulation

Let $G = (V, E)$ be an access graph and let us define, for each edge $\{u, v\} \in E$, its associated decision variable

$$x_{u,v} = \begin{cases} 1, & \text{if } \{u,v\} \text{ is selected} \\ 0, & \text{otherwise.} \end{cases}$$

For each variable (edge) $x_{u,v}$, let $w_{u,v}$ denote its associated weight. Then an integer programming formulation for the maximum-weight path cover problem can be stated as:

$$\begin{aligned}
\max \quad & \sum_{\{u,v\}\in E} w_{u,v} x_{u,v} & & & \\
\text{s.t.} \quad & \sum_{\{u,v\}\in E} x_{u,v} & \leq 2 & \quad \text{for all } v \in V & (5.1) \\
& \sum_{\{u,v\}\in C} x_{u,v} & \leq |C| - 1 & \quad \text{for all cycles } C \subseteq E & (5.2) \\
& x_{u,v} & \in \{0,1\} & \quad \text{for all } \{u,v\} \in E &
\end{aligned}$$

The objective function maximizes the total weight of the selected edges. The degree inequalities (5.1) enforce each vertex to have at most two incident edges in the cover. Their number is linear in $|V|$. The cycle inequalities (5.2) exclude any solutions that contain cycles from the feasible set. In general, if the edge-set E is not sparse (it could, e.g., be completely cycle-free), the number of possible cycles $C \subseteq E$ may be exponential in the size of the input data. This can be easily verified when considering the number of vertex-subsets $W \subsetneq V$, which is clearly exponential in $|V|$. For instance, in a complete graph, it is easy to create one or several cycles for each subset W of cardinality at least three.

5.1.1.1 Separation of the Cycle Inequalities

As discussed above, the number of cycle inequalities becomes impossible to be completely considered in a linear program for larger input sizes. We did not yet discuss how to separate them. Exploiting that there are exactly $|C|$ summands on the left hand side, we subtract this value from both sides. This turns inequality (5.2) into:

$$\sum_{\{u,v\}\in C}(x_{u,v}-1) \leq -1 \qquad \text{for all cycles } C \subseteq E$$

Now multiplying the inequalities with -1 yields the following form.

$$\sum_{\{u,v\}\in C}(1-x_{u,v}) \geq 1 \qquad \text{for all cycles } C \subseteq E \tag{5.3}$$

The last version is more suitable to perform an efficient separation algorithm by considering the so-called *support graph*. The support graph is equivalent to the instance graph in terms of its vertices and edges, but with all the edges having their LP values assigned as their weights. In this particular case, we however need to use the 'inverted' LP values $1 - x_{u,v}$ for each edge $\{u, v\} \in E$. Considering inequalities (5.3), it is easy to see that a cycle inequality is violated, if and only if we can find a pair of vertices $s \neq t \in V$ such that there is a cycle in the constructed support graph containing s and t of length strictly smaller than one. Conversely, there is *no* violated cycle inequality, if and only if *all shortest paths* for different pairs of vertices $s, t \in V$ together with their 'closing' edges $\{s, t\}$ do not admit such a cycle. The separation of the cycle inequalities can therefore be carried out by $|E|$ shortest path computations [GJR85b].

5.1.2 Hamiltonian Path Formulation

We rely on the same definitions and variable semantics as defined for the path cover formulation. However, we will now assume that the access graph $G = (V, E)$ has been completed by adding zero-weight edges. One way to formulate the maximum-weight Hamiltonian path problem as an integer program would then be the following:

$$\begin{aligned}
\max \quad & \sum_{\{u,v\}\in E} w_{u,v}x_{u,v} & & \\
\text{s.t.} \quad & \sum_{\{u,v\}\in E} x_{u,v} & \leq 2 \qquad & \text{for all } v \in V \\
& \sum_{\{u,v\}\in C} x_{u,v} & \leq |C|-1 \qquad & \text{for all cycles } C \subseteq E \\
& \sum_{\{u,v\}\in E} x_{u,v} & = |V|-1 \qquad & \\
& x_{u,v} & \in \{0,1\} \qquad & \text{for all } \{u,v\} \in E
\end{aligned} \tag{5.4}$$

The formulation just stated coincides with the path cover formulation except for the additional equation (5.4). It forces exactly $|V|-1$ edges to be selected which is a necessary condition to obtain a Hamiltonian path. The other inequalities and also the maximizing objective function do not already impose this condition. This is true especially for the case that G is not Hamiltonian without the artificially added zero-weight edges. Then, without the additional equation, any path cover would be a feasible solution and, the maximally weighted among them, would be even optimum solutions.

To gain more advantages from the transformation, we strive to replace the cycle inequalities by the following vertex-set-oriented inequalities.

$$x(E(W)) \leq |W|-1 \qquad \text{for all } W \subseteq V,\ |W| \geq 2 \tag{5.5}$$

Here, $E(W)$ is the subset of E consisting of all the edges with both endpoints in the set W, i.e., $E(W) = \{\{u,v\} \in E \mid u,v \in W\}$. Straightforwardly, $x(E(W)) = \sum_{\{u,v\}\in E(W)} x_{u,v}$ is then the sum of the variables associated to $E(W)$. Inequalities (5.5) exclude *all* possible cycles associated with each vertex set $W \subseteq V$ from the set of feasible solutions [DFJ54]. This is a stronger property than achieved with the cycle inequalities that forbid exactly one cycle per inequality. There is also symmetry in that if the inequality for $W \subseteq V$ is satisfied, then it is also satisfied for $V \setminus W$ and vice versa. It is a well-known result [Hon72] from investigations of the Traveling Salesman Problem (TSP) that inequalities (5.5) can be separated in polynomial time. To use the corresponding separation procedure in practice, it is however necessary to turn the degree inequalities into equations, i.e., to enforce each vertex to have exactly two adjacent vertices in a feasible solution. Under these preconditions, inequalities (5.5) are equivalent to

$$x(\delta(W)) \geq 2 \qquad \text{for all } W \subseteq V,\ |W| \geq 2 \tag{5.6}$$

where $\delta(W) = \{\{u,v\} \in E \mid u \in W \text{ and } v \notin W\}$ [DFJ54]. The violation of inequalities (5.6) by LP solutions can now be tested by a minimum-cut computation using the support graph. Looking at the inequalities, it also becomes clear why it is necessary to have degree equations instead of inequalities. Let $P \subset E$ be a Hamiltonian path and let $v \in V$ be some end vertex of P that has thus degree one. Further, let u be adjacent to v in P. Then, inequality (5.6) is violated for $W = \{u,v\}$, since the only edge in $\delta(W)$ is the one leaving u to its other neighbor. It is not possible to exclude the end vertices from the constraints, since it is of course not known a priori which vertices will be end vertices of an optimum Hamiltonian path. Hence, to circumvent this problem, we need to make sure that there is no vertex in a solution that has a degree different from two. By replacing the degree inequalities by equations, we have however turned the Hamiltonian path problem into a Hamiltonian cycle problem. This has two major consequences. Firstly, we need to allow cycles of full length, i.e., visiting all the graph's vertices. This can be done by restricting inequalities (5.6) to proper subsets W of V. They are then called *subtour elimination constraints* (SECs). Secondly, we need to add an additional vertex z to the graph, and to connect it to all original vertices using zero-weight edges. The vertex z then serves as a cutting point for the cycle in order to obtain the Hamiltonian path that we really want to compute, like illustrated in Fig. 5.1.

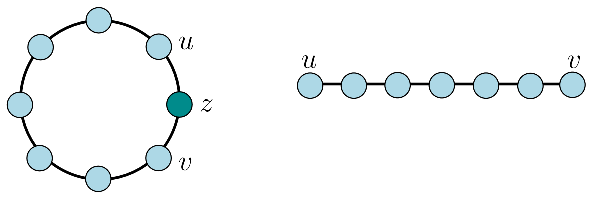


Figure 5.1: The role of the additional vertex z when turning a cycle into a path.

Without such a vertex, the cycle would have to be 'closed' using an original edge. Since its weight would contribute to the objective function, this condition could lead to different optima. It is misleading and not sufficient to simply remove the edge with the smallest weight from a so computed cycle, since there might be a path with larger total weight provided that there is no need to construct a cycle from it. An example is shown in Fig. 5.2. Assuming $M > 0$, the four edges with weight M form an optimal Hamiltonian cycle. Removing a minimum-weight edge results in an Hamiltonian Path of length $3M$. Selecting only two of the M-edges (dash-dotted) permits to use the edge with weight $M+1$. The cycle must then be closed using the zero-weight edge. However, after removing it again, we have obtained a Hamiltonian path of length $3M + 1$.

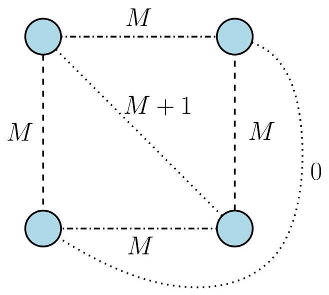


Figure 5.2: A simple K_4-counterexample that shows that optimal Hamiltonian paths cannot always be constructed from Hamiltonian cycles if no additional vertex is present. The Hamiltonian path consisting of the dotted and dash-dotted edges is superior to any Hamiltonian path constructed by removing an edge from the optimal Hamiltonian cycle (dashed and dash-dotted edges).

From now on, we assume that the vertex set V represents the program variables $\mathcal{V}$ and comprises the mentioned additional vertex z. Similarly, the complete edge set E is assumed to be appended by the zero-weight edges connecting z to all the original vertices. The corresponding Hamiltonian cycle based integer programming formulation is then:

$$\begin{aligned}
\max \quad & \sum_{\{u,v\}\in E} w_{u,v} x_{u,v} & & \\
\text{s.t.} \quad & \sum_{\{u,v\}\in E} x_{u,v} & = 2 \qquad & \text{for all } v \in V \\
& x(\delta(W)) & \geq 2 \qquad & \text{for all } W \subsetneq V,\ |W| \geq 2 \\
& x_{u,v} & \in \{0,1\} \qquad & \text{for all } \{u,v\} \in E
\end{aligned} \tag{5.7}$$

This formulation is equivalent to the associated standard formulation of the TSP, except for the objective function. It has been already mentioned, that the subtour elimination constraints provide stronger restrictions than the cycle inequalities. Besides that, the transformation into a Hamiltonian cycle problem permits the application of several strong inequalities known for the TSP. This, in turn, allows for an improved separation of fractional LP solutions and might yield much better upper bounds on the objective function. For a discussion of the many valid and facet-defining inequalities for the TSP, the interested reader is kindly referred to the vast pertinent literature, with [GP79a, GP79b, GP85, PR90, ABCC06] being potential starting points. For our Hamiltonian cycle based solver implementations, we only consider one additional class of facet-defining inequalities, namely the *two-matching inequalities* [Edm65]. Using the same notation as for the SECs, they can be written as the inequalities

$$x(E(H)) + x(T) \leq |H| + \frac{1}{2}(|T| - 1) \qquad \text{for all } H \subsetneq V \text{ and for all } T \subsetneq E \tag{5.8}$$

where H (typically called 'the handle') and T ('teeth') satisfy the conditions

(1) $|e \cap H| = 1$ for all $e \in T$ (all teeth share exactly one vertex with the handle),

(2) all edges in T are vertex-disjoint,

(3) and $|T| \geq 3$ and odd.

In our implementations, we use an exact and polynomial-time separation procedure that has been proposed by Padberg and Rao [PR82], see also [PR90].

5.1.2.1 A Short Note on the Trade-Off Between Both Formulations

The main advantage of the path cover formulation is its smaller size for sparse access graphs while the main advantage of the Hamiltonian path formulation is its stronger separation potential due to the subtour elimination constraints.

If strong upper bounds are needed in order to prove optimality of a known feasible solution, then the Hamiltonian path formulation will typically perform better. However, for smaller instances where the optimum objective function value nearly matches the upper bound provided by the initial linear programming relaxation (without any cycle inequalities and SECs respectively), the path cover variant may well be faster.

5.2 General Offset Assignment

To solve GOA to global optimality, we need to solve the two interdependent subproblems described in Sect. 4.3 and 4.4 in an integrated fashion, i.e., we need to find a memory layout that will allow us to create the best possible address register assignment.

A key observation is that the objective function is the only point where the memory layout influences the concrete ARA network problem to be solved. The cost of an access transition (u, v) in the network described in Sect. 4.4 is zero if and only if the variables associated with u and v are equal or neighbors in the memory layout. Otherwise, a positive cost c_A reflecting the overhead of an additional address arithmetic instruction is assigned. Moreover, there is no reason to not redefine this rule for $r > 1$, i.e., to assign arcs $(u, v) \in A$ the cost zero if u and v are no more than r positions apart from each other and c_A otherwise. However, in terms of modeling the feasible solutions of GOA problems, it makes a considerable difference whether r is equal to one or may be larger. We will now first discuss the approach for $r = 1$ and then proceed to the more general case.

5.2.1 Models Supporting Autoin-/decrement Instructions

For $r = 1$, the situation is similar to the one discussed in Sect. 5.1. Concerning the memory layout, we have basically the same modeling opportunities as in the SOA case. To avoid redundancy in the following descriptions, we restrict ourselves to Hamiltonian cycle based formulations as it should be clear from Sect. 5.1 how to derive the one-to-one-corresponding path cover variants from them. One issue however should be mentioned: In order to use the path cover formulation for $k > 1$, one may have to extend the access graph by zero-weight edges. It is not necessarily required to make it complete like in the Hamiltonian path version, however, there has to be a variable $x_{u,v}$ in the integer program for every pair of program variables u, v such that u precedes v in the access sequence. This is necessary because $x_{u,v}$ will be used to define the cost of the associated access transition (u, v) as is explained in the following.

To model GOA completely, we will now always consider two graphs. Firstly, a complete graph $G = (V, E)$ with a vertex for each of the program variables $\mathcal{V}$ and the additional vertex $z \in V$ as described in Sect. 5.1.2. Secondly, we have a network $N = (V_N, A)$ with $V_N = V_S \cup \{s, t\}$ where V_S is a vertex set related to the accesses contained in the input sequence S, just like in Sect. 4.4. Let $A_S = \{(v, w) \mid v, w \in V_S,\ v < w\}$ and $A = A_S \cup \{(s, v) \mid v \in V_S\} \cup \{(v, t) \mid v \in V_S\}$. Since all access vertices in V_S are instances of the program variables $\mathcal{V}$ represented by vertices of the set V, we may define a corresponding unique mapping $\sigma : V_S \to V$. For ease of reference, we further split the set A_S into $A_S^{\neq} = \{(u, v) \in A_S,\ \sigma(u) \neq \sigma(v)\}$, i.e., the set of arcs between accesses that do not refer to the same associated program variable and, analogously, the set $A_S^{=}$.

In addition to the flow arc variables $y_{u,v}$ for each arc $(u,v) \in A$, we associate edge decision variables $x_{u,v} \in \{0,1\}$ with the edges $\{u,v\} \in E$ that have no associated costs. The variable $x_{u,v}$ is equal to one if the edge $\{u,v\}$ is part of the computed Hamiltonian cycle (u and v are neighbors in the memory layout), and zero otherwise. Since G is undirected, the variables $x_{u,v}$ are by convention only defined for $u < v$. Slightly disregarding mathematical precision, we write $x_{\sigma(u),\sigma(v)}$ when referring to the associated edge decision variable of $y_{u,v}, (u,v) \in A_S^{\neq}$, irrespective of whether $\sigma(u) < \sigma(v)$ or $\sigma(u) > \sigma(v)$. Exploiting these variable relationships, we can express the cost of an access transition $y_{u,v}$ with $(u,v) \in A_S^{\neq}$ by $(1 - x_{\sigma(u),\sigma(v)})c_A$ while the cost of each variable $y_{u,v}$ for $(u,v) \in A_S^{=}$ is zero. This leads to a first quadratic integer programming formulation for GOA.

5.2.1.1 Quadratic Formulation

$$\begin{array}{llll}
\min & \displaystyle\sum_{(u,v)\in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)}) c_A y_{u,v} + \sum_{v \in V_S} c_L y_{s,v} & & \\
\text{s.t.} & \displaystyle\sum_{\{u,v\}\in E} x_{u,v} & = 2 & \text{for all } v \in V \\
& x(\delta(W)) & \geq 2 & \text{for all } W \subsetneq V,\ |W| \geq 2 \\
& \displaystyle\sum_{(u,v)\in A} y_{u,v} & = 1 & \text{for all } u \in V_S \\
& \displaystyle\sum_{(u,v)\in A} y_{u,v} & = 1 & \text{for all } v \in V_S \\
& \displaystyle\sum_{v\in V_S} y_{s,v} & \leq k & \\
& x_{u,v} & \in \{0,1\} & \text{for all } (u,v) \in E \\
& y_{u,v} & \in \{0,1\} & \text{for all } (u,v) \in A
\end{array}$$

This integer program is essentially the min-cost flow formulation from Sect. 4.4 appended by inequalities from Sect. 5.1.2 enforcing the edge variables to correspond to a Hamiltonian cycle of G and with a new objective function linking the two subproblems. The objective function simply sums up the terms $(1 - x_{\sigma(u),\sigma(v)})c_A$ for all arcs $(u,v) \in A_S^{\neq}$ and all costs $\sum_{v \in V_S} c_L y_{s,v}$ for initial address register loads.

The above integer program is quadratic in its objective function. We may linearize it using the standard linearization approach. However, first we simplify. The term

$$\min \sum_{(u,v)\in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)}) c_A y_{u,v}$$

can also be written as

$$\min \Big(\sum_{(u,v)\in A_S^{\neq}} y_{u,v} - \sum_{(u,v)\in A_S^{\neq}} x_{\sigma(u),\sigma(v)} y_{u,v} \Big) c_A.$$

Following Sect. 1.6, we then need $|A_S^{\neq}|$ new variables $z_{u,v} = x_{\sigma(u),\sigma(v)} y_{u,v}$ and three linearization constraints for each of the new variables:

$$\begin{aligned}
z_{u,v} &\leq x_{\sigma(u),\sigma(v)} \\
z_{u,v} &\leq y_{u,v} \\
z_{u,v} &\geq x_{\sigma(u),\sigma(v)} + y_{u,v} - 1
\end{aligned}$$

After this transformation, the objective function becomes:

$$\min \sum_{(u,v)\in A_S^{\neq}} c_A y_{u,v} - \sum_{(u,v)\in A_S^{\neq}} c_A z_{u,v} + \sum_{v\in V_S} c_L y_{s,v}$$

Clearly, the number of product variables to be introduced, $|A_S^{\neq}|$, must be strictly smaller than the total number of flow arc variables which is $(|S|+2)\cdot(|S|+1)/2$. Hence, in total the linearized version of the above formulation has strictly less than $|\mathcal{V}|\cdot(|\mathcal{V}|-1)/2 + ((|S|+2)\cdot(|S|+1)/2)$, i.e., $\mathcal{O}(|\mathcal{V}|^2+|S|^2)$ variables. As already discussed in Sect. 5.1.2, the number of subtour elimination constraints is exponential in $|\mathcal{V}|$ such that it is preferable to not consider all of them from the beginning. The number of remaining nontrivial constraints is strictly less than $1+|\mathcal{V}|+2(|S|+2)+3((|S|+2)\cdot(|S|+1)/2)$, i.e., $\mathcal{O}(|\mathcal{V}|+|S|^2)$.

Remarkably, due to the properties of the min-cost flow part addressed in Sect. 4.4.4, all these numbers are independent from the number k of ARs available. The more access pairs in S refer to the same variable, the less product variables and associated constraints are needed.

5.2.1.2 Linear Formulation

By further inspection and by exploiting the fact that there are only two cases for each arc $(u,v) \in A_S^{\neq}$, namely that it either has the assigned cost c_A or assigned cost zero, the problem can be linearized inherently, that is without generating any products that need a subsequent linearization. The main idea is to replace every variable (arc) between two accesses $y_{u,v}$, $(u,v)\in A_S^{\neq}$ by two new variables (arcs) $y_{u,v}^0$ and $y_{u,v}^c$ reflecting the two mentioned cases (cf. Fig. 5.3). The set $A_S^{\neq}$ is therefore once more split into the corresponding new arc sets A_S^0 and A_S^c. For every arc $(u,v)\in A_S^{=}$, we keep the former variable $y_{u,v}$ with zero cost as before. We also skip the superscript when referring to flow variables disregarding their costs or if only one instance exists.

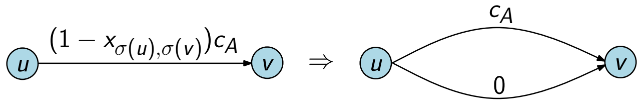


Figure 5.3: Replacing arcs with dynamic costs by two arcs with static costs.

The new network N has now the arc set $A = A_S^0 \cup A_S^c \cup A_S^{=} \cup \{(s,v) \mid v\in V_S\} \cup \{(v,t) \mid v\in V_S\}$. The new objective is of course to minimize the selected arcs with positive costs assigned. This is a linear expression in the set of variables. However, we now have to restrict the use of zero-cost arcs. As before in the quadratic model, it should only be possible to use them if the corresponding variables are neighbors in the access sequence. With the newly introduced variables this can easily be enforced using the following constraints:

$$y^0_{u,v} \leq x_{\sigma(u)\sigma(v)} \qquad \text{for all } (u,v) \in A^0_S$$

The following constraint is also valid for the model:

$$y^c_{u,v} \leq 1 - x_{\sigma(u)\sigma(v)} \qquad \text{for all } (u,v) \in A^c_S$$

However, since zero-cost arcs are preferred by the objective function, there will never be a variable $y^c_{u,v} = 1$ in an optimum solution where also $x_{\sigma(u)\sigma(v)} = 1$, even if these constraints are not present. Therefore, this constraint will also have only marginal impact on the solution process and can be omitted.

A complete linear IP formulation for GOA with $r = 1$ is then:

$$\begin{array}{llll}
\min & \sum\limits_{(u,v)\in A^c_S} c_A y^c_{u,v} + \sum\limits_{v\in V_S} c_L y_{s,v} & & \\
\text{s.t.} & \sum\limits_{\{u,v\}\in E} x_{u,v} & = 2 & \text{for all } v \in V \\
& x(\delta(W)) & \geq 2 & \text{for all } W \subsetneq V,\ |W| \geq 2 \\
& \sum\limits_{(u,v)\in A} y_{u,v} & = 1 & \text{for all } u \in V_S \\
& \sum\limits_{(u,v)\in A} y_{u,v} & = 1 & \text{for all } v \in V_S \\
& \sum\limits_{v\in V_S} y_{s,v} & \leq k & \\
& y^0_{u,v} & \leq x_{\sigma(u)\sigma(v)} & \text{for all } (u,v) \in A^0_S \\
& x_{u,v} & \in \{0,1\} & \text{for all } (u,v) \in E \\
& y_{u,v} & \in \{0,1\} & \text{for all } (u,v) \in A
\end{array}$$

The number of variables is the same as in the quadratic formulation since essentially for each $(u,v) \in A^{\neq}_S$ the product variable $z_{u,v}$ is replaced by a second flow arc variable $y^c_{u,v}$. However, $2|A^{\neq}_S|$ less constraints are needed.

5.2.2 Models Supporting General Auto-Modify Instructions

If the processor supports offset ranges r strictly larger than one, the mathematical modeling of the associated optimization problem becomes much more complicated as has already been discussed in Sect. 4.3.2. The only straightforward way to model the problem correctly appears to be via using assignment variables that encode the position of the program variables explicitly. We will now develop a formulation that has considerably less variables and constraints than previously published assignment-based formulations [WG97, OKT06] while preserving the advantages of our model for $r = 1$.

In the assignment problem, we have variables $x_{i,p}$ that take value one if item i is placed at position p and zero otherwise. To model the stack memory layout for $n = |\mathcal{V}|$ variables, we need exactly n^2 variables, since any variable may be placed at any position $p \in P, P = \{1, \dots, n\}$. Clearly, every variable $v \in \mathcal{V}$ must be assigned exactly one position $p \in P$ and each position $p \in P$ must be assigned exactly one variable $v \in \mathcal{V}$. Hence, the corresponding constraints of the assignment problem are:

$$\sum_{p \in P} x_{v,p} = 1 \qquad \text{for all } v \in \mathcal{V}$$

$$\sum_{v \in \mathcal{V}} x_{v,p} = 1 \qquad \text{for all } p \in P$$

Let $x_{u,a} = 1$ and $x_{v,b} = 1$ with $u \neq v$ and $a \neq b$. Then, an access transition $u \to v$ has cost zero if and only if $|b-a| \leq r$. For the following discussion, we introduce auxiliary variables $r_{u,v}$ for each pair of different variables $u, v \in \mathcal{V}$, expressing whether u and v are placed within range r or not. We will however not need these variables for the subsequently developed integer program. With the auxiliary variables at hand, we may express the following constraints:

$$r_{u,v} \geq x_{u,b} + x_{v,a} - 1 \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| \leq r$$

$$r_{u,v} \geq x_{u,a} + x_{v,b} - 1 \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| \leq r$$

These inequalities force $r_{u,v}$ to become one as soon as the assignment of positions to u and v is such that their distance is at most r. Further, they never enforce $r_{u,v}$ to be greater than one. We may also directly combine the constraints as follows:

$$r_{u,v} \geq \underbrace{x_{u,b} + x_{u,a}}_{\leq 1} + \underbrace{x_{v,a} + x_{v,b}}_{\leq 1} - 1 \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| \leq r \tag{5.9}$$

On the other hand, we must make sure that $r_{u,v}$ is never assigned value one if the two variables are not placed within range r using the following constraints:

$$r_{u,v} \leq 2 - x_{u,b} - x_{u,a} - x_{v,a} - x_{v,b} \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } |b-a| > r \tag{5.10}$$

In total, this would already amount to $\binom{n}{2}\binom{n}{2}$ constraints, having not yet formulated constraints that restrict the use of the zero-cost flow arc variables. However, we can still improve on that. First of all, for the same reason as in the linear IP presented in Sect. 5.2.1.2, we do not need to care about the positive cases that permit to use a flow arc variable $y^0_{u,v}$ but only forbid those cases where the use of

$y^0_{u,v}$ is prohibited. Hence, we can completely omit the constraints (5.9). Now, we take a closer look on constraints (5.10) again, keeping in mind that each variable $v \in \mathcal{V}$ can be assigned at most one position from any proper subset Q of P, i.e., $\sum_{p \in Q} x_{v,p} \leq 1$ for all $v \in \mathcal{V}$ and $Q \subset P$. Say variable u is fixed at position a, then all the positions of v that make the transition $u \to v$ have nonzero cost are the positions $p \in [1, a-r-1]$ and $p \in [a+r+1, n]$. Hence, by fixing one position, we can reformulate (5.10) by:

$$r_{u,v} \leq \; 2 - x_{u,a} - \underbrace{\sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p}}_{\leq 1} \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a \in P$$

Since we now exactly characterized under which conditions two variables u and v are not within range r, we can again exploit this to apply the correct restriction on the use of each zero-cost flow arc variable $y^0_{u,v}$:

$$y^0_{u,v} \leq \; 2 - x_{u,a} - \sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p} \qquad \text{for all } (u,v) \in A^0_S \text{ and } a \in P \tag{5.11}$$

For any of the at most $\binom{|S|}{2}$ variables $y^0_{u,v}$, there are at most n positions where u can be fixed at, so we obtain only $\mathcal{O}(|S|^2 \cdot |\mathcal{V}|)$ constraints (5.11) in total. The full IP formulation is then:

$$\begin{array}{lll}
\min & \displaystyle\sum_{(u,v) \in A^c_S} c_A y^c_{u,v} + \sum_{v \in V_S} c_L y_{s,v} & \\
\text{s.t.} & \displaystyle\sum_{p \in P} x_{v,p} \quad = 1 & \text{for all } v \in \mathcal{V} \\
& \displaystyle\sum_{v \in \mathcal{V}} x_{v,p} \quad = 1 & \text{for all } p \in P \\
& \displaystyle\sum_{(u,v) \in A} y_{u,v} \quad = 1 & \text{for all } u \in V_S \\
& \displaystyle\sum_{(u,v) \in A} y_{u,v} \quad = 1 & \text{for all } v \in V_S \\
& \displaystyle\sum_{v \in V_S} y_{s,v} \quad \leq k & \\
& y^0_{u,v} \quad \leq 2 - x_{u,a} - \displaystyle\sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p} & \text{for all } (u,v) \in A^0_S \text{ and } a \in P \\
& x_{v,p} \quad \in \{0,1\} & \text{for all } v \in \mathcal{V} \text{ and } p \in P \\
& y_{u,v} \quad \in \{0,1\} & \text{for all } (u,v) \in A
\end{array}$$

Being more precise, the model has $|\mathcal{V}|^2 + ((|S|+2) \cdot (|S|+1)/2)$, i.e., again $\mathcal{O}(|\mathcal{V}|^2 + |S|^2)$ variables. The number of nontrivial constraints is bounded from above by $1 + 2|\mathcal{V}| + 2(|S|+2) + ((|S|+2) \cdot (|S|+1)/2) \cdot |\mathcal{V}|$, i.e., $\mathcal{O}(|\mathcal{V}| \cdot |S|^2)$.

5.3 Branch-and-Cut Implementations

The preceding sections give rise to several solver implementations. The following list provides an overview of the different implemented branch-and-cut algorithms.

- `SOA-HC`: The SOA solver based on Hamiltonian cycle computations.
- `SOA-PC`: The SOA solver based on path cover computations.
- `GOA-HC`: The inherently linear GOA solver based on Hamiltonian cycle computations.
- `GOA-PC`: The inherently linear GOA solver based on path cover computations.
- `GOA-QHC`: The linearized (originally quadratic) GOA solver based on Hamiltonian cycle computations.
- `GOA-QPC`: The linearized (originally quadratic) GOA solver based on path cover computations.
- `GOA-ASS`: The linear GOA solver based on the assignment problem formulation.

The solvers adhere to the general branch-and-cut scheme as described in Sect. 1.5, relaxing integrality and the respective classes of inequalities that shall be separated. We report on the solver-specific details in terms of cutting plane generation and primal heuristics in the subsequent sections.

Unlike in the case of the scheduling solver implementation (see the discussion in Sect. 3.6.6, it has been found acceptable to implement and test the presented offset assignment models using the commercial IP solver CPLEX 12.6 [CPL13], aware of the fact that the internal mechanisms of a commercial tool may contribute to their sustained performance. For instance, CPLEX may decide to separate further general cutting planes for integer programs and uses a set of internal heuristics, e.g., to select the variables to branch on. We disabled internal presolving techniques that are not compatible with the application of cutting planes but adopted all other default parameters.

The difference compared to the situation in the scheduling chapter is that, in the offset assignment context, the models presented are the very first methods capable to solve a larger set of instances at all. In this sense, there is no danger that they appear superior to the rare other existing methods just by means of some black-box features. In addition, the following experiments do not primarily aim at providing running time results but are also an evaluation of the impact of certain processor features on the overall address computation overhead, and of the quality of heuristic solutions. For these purposes, it is reasonable to look at the implementations rather as an assessment of the practical applicability of the underlying concepts with emphasis on exploring how far one can go if fast branch-and-cut frameworks are at hand. This is sensible as well in terms of obtaining a maximum number of optimal solutions to the reference benchmark instances, permitting a maximum number of comparisons to heuristic solutions.

5.3.1 Cutting Plane Generation

For the path-cover-based solvers, the cycle inequalities are separated as described in Sect. 5.1.1.1. In case of the Hamiltonian-cycle-based solvers, the separation routine for subtour elimination constraints is called first. If it finds violated inequalities, these are added to the LP and the separation procedure terminates. If it does not find any cuts and the LP solution has fractional components, the exact separation procedure for two-matching-inequalities [PR82] is invoked.

We also decided to separate inequalities (5.11) from Sect. 5.2.2 in the assignment-based solver implementation. Although their number is polynomial in the size of the input data, these inequalities quickly become a limitation for larger instances due to increased LP solution times. Moreover, typically only a fraction of them is in fact required (i.e., ever violated) during the solution process.

5.3.2 Primal Heuristics

The implemented methods follow the general idea that variables with an LP value close to one are likely to be part of a good or even optimal solution. In particular, we greedily construct a memory layout based on the LP solution. For the SOA solvers, this is already sufficient. For the GOA cases, we exploit the condition that we can compute an optimal ARA for a given memory layout relatively quickly in practice.

The pseudocodes that are listed as Alg. 4-6 illustrate the procedures. In case of the path cover and Hamiltonian-cycle-based solvers, we iteratively select feasible edges $\{u, v\}$ in nonincreasing order of the LP values of their corresponding variables $x_{u,v}$ as long as this is feasible. More precisely, for the SOA variants, we use the LP values multiplied with the respective cost coefficients. There is nearly no difference between the two associated procedures since we, also in the path cover version, simulate a complete graph and assume negative LP values for all the truly nonexisting edges that only serve for concatenations. This artificially moves them to the end of the sorted array. For the assignment-based solver, we assign each program variable $v \in \mathcal{V}$ the free position that is mostly preferred by its assignment variables $x_{v,p}, p \in \{1, \ldots, n\}$. After constructing an offset assignment like this, the network flow problem from Sect. 4.4.2 is solved to find an optimal ARA.

5.4 Heuristics

Since quality results of several heuristics in relation to optimum solutions have been published already in [JM13, ML14, Mal14] and the emphasis in this thesis is on the exact approaches, only a few heuristics are incorporated in the following experiments. They are selected in such a way that it is possible to give an impression of how large the gaps of well-performing heuristic methods on the one hand, and naive ones on the other, are in practice. A naive approach to SOA is to construct a memory layout simply by the order of first use (OFU) of the program variables. We will denote such a method by `SOA-OFU`. The best performing SOA heuristic in [Leu03, JM13] is

Algorithm 4 Primal Heuristic Framework

```
function PRIMALHEURISTIC($x$, $N = (V_N, A)$)
    $OA \leftarrow$ COMPUTEOFFSETASSIGNMENT($x$)
    SETCOSTS($N$, $OA$)                  # Set arc costs based on distances in OA
    $ARA \leftarrow$ MINCOSTFLOW($N$)
```

Algorithm 5 Memory Layout Subroutine (Hamiltonian Cycle/Path Cover Version)

```
function COMPUTEOFFSETASSIGNMENT($x$)
    $G = (V, E)$                                                # Complete graph
    SORT($E, x$)                 # Sort edges nonincreasingly w.r.t. their LP values
    INITIALIZEUNIONFIND($V$)
    $n \leftarrow |V|$, $m \leftarrow |E|$
    $select \leftarrow \emptyset$, $count \leftarrow 0$
    for $i = 1 \rightarrow n$ do
        $deg(i) \leftarrow 0$                               # Initialize degrees to zero
    for $i = 1 \rightarrow m$ do
        $e = \{u, v\} \leftarrow E[i]$
        if $deg(u) < 2$, $deg(v) < 2$, $count < n - 1$ and (FIND($u$) $\neq$ FIND($v$)) then
            $select \leftarrow select \cup \{e\}$, $count \leftarrow count + 1$
            $deg(u) \leftarrow deg(u) + 1$, $deg(v) \leftarrow deg(v) + 1$
            UNION($u, v$)
    $OA \leftarrow$ CREATEPATH($select$)                    # Join edges at common vertices
    return $OA$
```

Algorithm 6 Memory Layout Subroutine (Assignment Version)

```
function COMPUTEOFFSETASSIGNMENT($x$)
    $n \leftarrow |\mathcal{V}|$
    for $p = 1 \rightarrow n$ do
        $pos[p] \leftarrow$ free     # Initialize all positions $p \in P = \{1, \ldots, n\}$ to be unassigned
    $X_{max} \leftarrow$ Array of LP-values $\max\{x_{v,p} \mid p \in \{1, \ldots, n\}\}$ for each $v \in \mathcal{V}$
    SORT($\mathcal{V}, X_{max}$)              # Sort variables $\mathcal{V}$ nonincreasingly w.r.t. $X_{max}$
    for $i = 1 \rightarrow n$ do
        $v \leftarrow \mathcal{V}[i]$
        $X_v \leftarrow$ Array of LP-values $x_{v,p}$ for each $p \in \{1, \ldots, n\}$.
        SORT($P, X_v$)                   # Sort positions $p \in P$ nonincreasingly w.r.t $X_v$
        for $j = 1 \rightarrow n$ do
            $p \leftarrow P[j]$
            if $pos[p]$ is free then
                $pos[p] = v$
                $OA[v] = p$
    return $OA$
```

called `SOA-INC-TB`. It combines an incremental method by Atri et al. [ARK01] with a tie-breaking that was proposed by Leupers and Marwedel [LM96].

For GOA, it turned out that it is typically more suggestive to first set up a memory layout and to compute then an optimal ARA for this layout instead to partition the problem into multiple SOA subproblems [ML14]. For $r > 1$, this appears to be even more advisable since most of the existing SOA algorithms used as subroutines are designed for $r = 1$ (they iteratively select edges) and it is not trivial to generalize them. In contrast to that, Gebotys' network model is easy to adapt for arbitrary auto-modify ranges as already discussed in Sect. 5.2 and also exploited by our primal heuristics. We therefore combine the optimal ARA approach with the two above mentioned heuristics to compute (SOA) memory layouts. The algorithms will respectively be referred to as `GOA-OFU-MCF` and `GOA-ITB-MCF`. Called with $k = 1$, they produce the same results as their SOA counterparts.

5.5 Experimental Results

5.5.1 OffsetStone

The following experiments are carried out with the *OffsetStone* benchmark set that has been extracted from 31 real-world application programs written in ANSI C. Among them are computationally intensive ones (e.g., audio, video and image compression, Fourier transformation) as well as control-dominated applications (e.g., gzip). It has been frequently used in publications dealing with offset assignment and therefore allows for meaningful comparisons. For details on how the instances were extracted, we refer to the original paper [Leu03].

We considered all the $2,785$ instances that consist of at least three program variables. Some statistics about their distribution are given in Fig. 5.4 and Fig. 5.5. For experiments that are carried out only on subsets of the instances, statistical data about these will be given separately. The maximum number of variables occurring is $1,336$ and there is a longest access sequence of length $3,340$ with 678 variables. However, in general, the instances are such that there are multiple access sequences associated with one program variable set. Hence, in the GOA case, multiple min-cost flow subproblems need to be solved per instance, all referring to the same memory layout. In this sense, the *cumulated access sequence lengths* depicted in Fig. 5.5 and Fig. 5.7 are the values obtained by summing up all the sequence lengths that belong to the same instance. Further, OffsetStone comprises sequences that refer to disjoint subsets of program variables such that the instances can be decomposed. This has also been exploited within all (including the heuristic) implementations.

5.5.2 Test System

All experiments were run single-threaded with an Intel Core i7-3770T processor (2.5 GHz) on a Debian Linux system with 8 GB RAM, `g++` 4.7.2, and optimization level `-O2`. CPU running times are averaged over five runs.

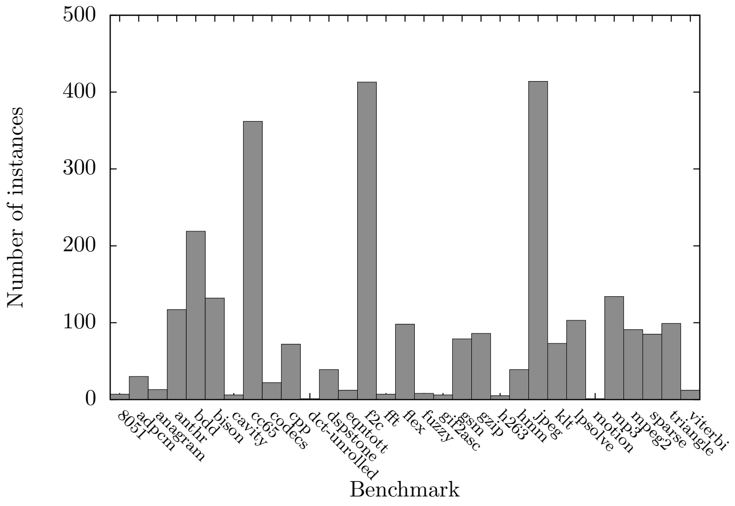


Figure 5.4: Distribution of instances among the benchmarks of OffsetStone.

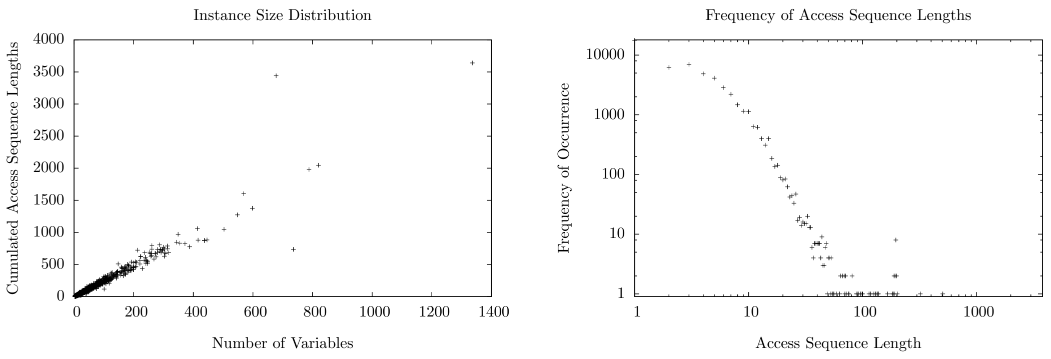


Figure 5.5: Distribution of the number of variables and access lengths across the instances.

5.5.3 Min-Cost Flow Implementation

For all minimum cost flow computations (by the primal heuristics used in the exact solvers as well as by the usual GOA heuristics), we called the network simplex algorithm [Orl96] provided by the LEMON C++ library [DJK11] in version 1.3. The asymptotic running time of the network simplex algorithm strongly depends on the used pivoting rule. We relied on the default block search rule of the library [KK12] that has a worst-case time bound of $\mathcal{O}(nm^2)$ with $n = |V_N|$ and $m = |A|$ (since, due to the assumption $c_A = c_L$, we have capacities and costs only zero and one), but typically performs much better in practice.

5.5.4 Simple Offset Assignment

We first consider the case $r = 1$ and compare the two SOA solver implementations with each other. On our test system, `SOA-HC` is able to solve all but two instances to optimality in less than ten seconds of CPU time (cf. Tab. 5.1). One of the instances, `mp3 86`, is the one with the largest number of variables and also a very long access sequence. The other one is also among the few largest instances contained in OffsetStone. Both can be solved by investing only a little more computation time. For a few instances, the separation of the two-matching inequalities was crucial for obtaining this performance. For example, the instance `motion 0` (with 280 program variables and a sequence of length 734) took 35.83 seconds without the additional cutting planes. However, in this very restrictive case with only a single register and only autoin-/decrements, they were not critical for whether an instance could or could not be solved at all.

			sequence lengths			
instance	**#vars**	**#seqs**	**min**	**max**	**sum**	`SOA-HC`
`cavity 0`	569	265	1	189	1,603	10.60
`mp3 86`	1,336	169	1	202	3,640	29.21

Table 5.1: The instances where `SOA-HC` timed out after ten seconds of CPU time.

In contrast to that, `SOA-PC` could not solve six instances within 60 seconds of CPU time (while most of them can be solved in less than a second by `SOA-HC`) and there were also three more instances, that needed more than ten seconds (cf. Tab. 5.2). This could be expected. For some rather small and 'easier' instances where the gap of the LP bounds and the quality of the first solutions found hardly differ, the path cover formulation is faster in many cases. However, when it comes to proving the optimality of found solutions in harder instances, the Hamiltonian-cycle-based formulations perform better since their cutting planes have a stronger impact on the bounds. Although the bound obtained with `SOA-PC` is also typically close to the optimum, it also fails more often in closing the gap entirely.

			sequence lengths			
instance	**#vars**	**#seqs**	**min**	**max**	**sum**	`SOA-PC`
`anthr 36`	133	29	1	43	326	10.33
`anthr 52`	414	95	1	37	1,059	> 60.00
`bdd 68`	221	62	1	42	508	> 60.00
`f2c 72`	290	218	1	19	739	> 60.00
`cavity 0`	569	265	1	189	1,603	14.49
`mp3 86`	1,336	169	1	202	3,640	> 60.00
`jpeg 293`	73	4	1	96	194	> 60.00
`jpeg 296`	147	70	1	130	395	12.37
`motion 0`	280	37	1	53	734	32.41
`mpeg2 15`	184	77	1	71	491	11.82
`mpeg2 90`	170	100	1	38	402	> 60.00

Table 5.2: The instances where `SOA-PC` timed out after ten seconds of CPU time.

Fig. 5.6 shows the average quality of `SOA-OFU` and `SOA-INC-TB` on all 2, 785 OffsetStone instances with at least three program variables. While the naive approach leads to a significant address computation overhead, the results of the greedy heuristic are already near-optimal. Considering single instances, we recognized maximal overheads for `SOA-INC-TB` of 12.5%. The `SOA-OFU` memory layouts however lead to sometimes more than twice explicit address arithmetic instructions as necessary. The SOA results presented here marginally differ from the presentation in [JM13] which is for two reasons. Firstly, the experiments there were carried out for instances with at least ten program variables. Secondly, the unavoidable cost for the first initialization of the address register was not counted there, leading to different quotients when computing the relative quality.

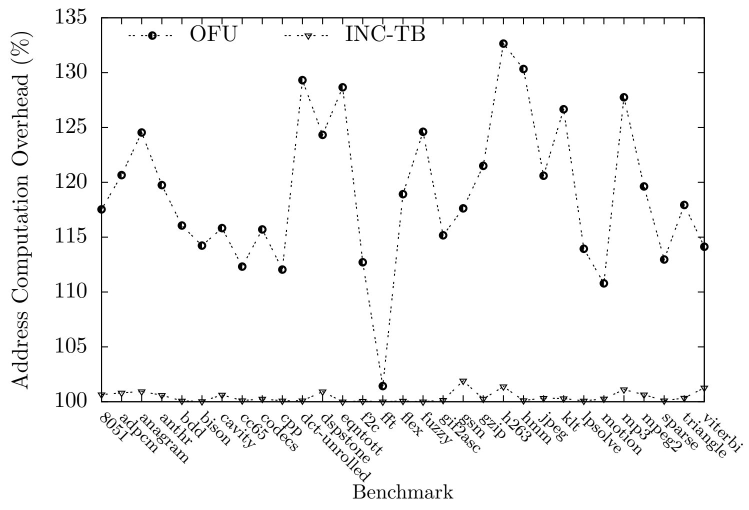


Figure 5.6: Average Quality of the selected SOA heuristics on OffsetStone.

It is of course possible to use the GOA solvers with $k = 1$ to solve the SOA problems. However, the complexity of the integer programs is higher due to the ARA subproblem parts and the linkage of the two subproblems in the objective function only will typically lead to weaker bounds than in the case of static cost coefficients. When `GOA-HC` (`GOA-QHC`) is used for the $k = 1$ case, eleven (thirteen) instances time out after ten seconds. With `GOA-PC` (`GOA-QPC`), thirteen (fifteen) instances time out. Each time, one further instance, `dct_unrolled 0`, fails to complete due to memory limitations. The instance has a single access sequence of length 3, 440 leading to a very large flow network with more than $\binom{3,442}{2}$ arc variables.

With a time limit of 60 seconds of CPU time, 2, 859 of the 2, 875 instances could be solved by all of the solvers except `GOA-ASS`. The total running times taken by these solvers as well as by the two heuristics are listed in Tab. 5.3. In total, the advantages of the Hamiltonian cycle formulation for more difficult instances more than compensate the slight disadvantages for the smaller ones.

SOA-OFU	SOA-INC-TB	SOA-HC	SOA-PC	GOA-HC	GOA-PC	GOA-QHC	GOA-QPC
1.47	1.97	37.87	79.19	85.29	80.92	146.38	149.54

Table 5.3: The total running times (in seconds) taken by various algorithms for 2, 859 instances.

5.5.5 General Offset Assignment

In the following experiment, we keep $r = 1$ fixed, but increase k. The precise numbers of timeouts (after ten seconds) of the exact GOA solvers (except GOA-ASS), are given in Tab. 5.4. Raising the time limit to 60 seconds again, we found that 2, 709 of the 2, 875 instances could be solved by all of them for all tested numbers of address registers. The total running times on these instances are given in Tab. 5.5. With increasing k, the relative performance of the path-cover-based and Hamiltonian-cycle-based solvers changes. While in the SOA case, the Hamiltonian cycle versions were typically slightly superior, GOA-PC has less timeouts within ten seconds and also better running times for $k \geq 2$ than GOA-HC. So in these cases, the advantages that the path-cover-based solvers gain from the sparsity of the OffsetStone instance (see also the discussion below) dominate the former advantage of the Hamiltonian-cycle-based solvers with their stronger separation opportunities. One can also see in Tab. 5.5 that the artificially linearized versions perform recognizably worse than their inherently linear counterparts. Remarkably, the number of timeouts and also the running times increase up to $k = 4$, but then slightly decrease again for $k = 8$. This however fits well with the quality results presented in Fig. 5.8, in that the total addressing cost for the OffsetStone instances can hardly be improved using more than four address registers. While the lower bounds provided by the LPs do typically not change significantly when increasing k, the primal heuristics have more opportunities (symmetric in their cost) to find good solutions. However, considered in relation, the differences are rather small and in any case (even for GOA-ASS), the numbers of timeouts increase only moderately with increasing k.

	GOA-HC	GOA-PC	GOA-QHC	GOA-QPC
$k = 1$	11 (0.40%)	13(0.47%)	13(0.47%)	15(0.54%)
$k = 2$	33 (1.19%)	25(0.90%)	54(1.94%)	35(1.26%)
$k = 4$	56 (2.01%)	46(1.65%)	95(3.41%)	78(2.80%)
$k = 8$	54 (1.94%)	45(1.62%)	86(3.09%)	76(2.73%)

Table 5.4: $r = 1$: Number of instances not solved by exact solvers after ten seconds.

	GOA-HC	GOA-PC	GOA-QHC	GOA-QPC
$k = 1$	40.10	31.73	52.74	43.49
$k = 2$	194.96	126.52	419.97	267.91
$k = 4$	276.39	161.88	981.50	572.77
$k = 8$	226.61	136.99	798.23	500.81

Table 5.5: $r = 1$: Total running times (in seconds) taken by all the exact GOA algorithms except GOA-ASS for 2, 709 instances.

GOA-ASS is not at all competitive to the other exact solvers for $r = 1$. With a time limit of ten seconds, it times out on $1,253$ instances (cf. Tab. 5.6). Its primal heuristic works well, but for many instances the basic constraints of the assignment problem part do not suffice in order to obtain lower bounds that prove optimality of known solutions. However, though its considerably worse performance compared to the other solvers, GOA-ASS solves already $10 - 20\%$ more instances for $r = 1$ than the implementation of the approach by Ozturk et al. in [ML14] and, except for the already mentioned instance dct_unrolled 0, it never failed to solve a problem due to memory limitations which was often the case for the latter.

We now consider not only autoin-/decrement instructions, but also larger auto-modify ranges. Since it can be assumed that the ranges are realized by reserved bits in the instruction opcode, we consider the cases $r = 1$, $r = 3$, and $r = 7$ associated to two, three and four bits respectively (each time with one sign bit). With increasing auto-modify ranges, GOA-ASS performs better which can be similarly explained as its bad performance for $r = 1$. With increasing r, the concrete memory layout becomes less important for an optimal address register assignment since more arcs in the min-cost flow network can be used without cost in any case. This effect is even stronger if the access sequence lengths are rather small which is often the case in OffsetStone as can be seen in Fig. 5.5. Hence, the lower bounds obtained from the LP and the upper bounds obtained by solutions found by the primal heuristics are much closer to each other and optimality of the latter can be proven much more often.

GOA-ASS	$r = 1$	$r = 3$	$r = 7$
$k = 1$	$1,253$ (44.99%)	924 (33.18%)	472 (16.95%)
$k = 2$	$1,245$ (44.70%)	880 (31.60%)	471 (16.91%)
$k = 4$	$1,246$ (44.74%)	888 (31.89%)	477 (17.13%)
$k = 8$	$1,252$ (44.96%)	881 (31.63%)	477 (17.13%)

Table 5.6: Number of instances not solved by GOA-ASS after ten seconds.

We identified $1,480$ instances with up to 200 program variables that GOA-ASS could solve to optimality for all tested choices of k and r within the time limit of ten seconds. If an exact solver for arbitrary r is desired for practical application, GOA-ASS could be improved by several means, e.g., by adding additional cutting planes that are valid for (quadratic) assignment problems. In general, the quadratic nature of the problem and its interdependent structure of two subproblems suggests a reformulation as a semidefinite program or the application of Bender's decomposition approach. The latter is even more suggestive due to the fact that the min-cost-flow subproblem is polynomial-time solvable. In this sense, GOA-ASS is to be seen as a 'proof of concept' to produce first results for $r > 1$ that allows to evaluate heuristics and the effect of exploitation of larger auto-modify ranges on the quality of address code generation.

Investing more computation time, we derived optimal solutions for $1,918$ of the $2,785$ instances for all mentioned combinations of r and k. For these instances, we display again the distribution of the number of variables and access sequence lengths in Fig. 5.7. As already mentioned, the number of longer access sequences is rather small which is one of the reasons why the instances become ‘easier’ or faster to solve for `GOA-ASS` with increasing r.

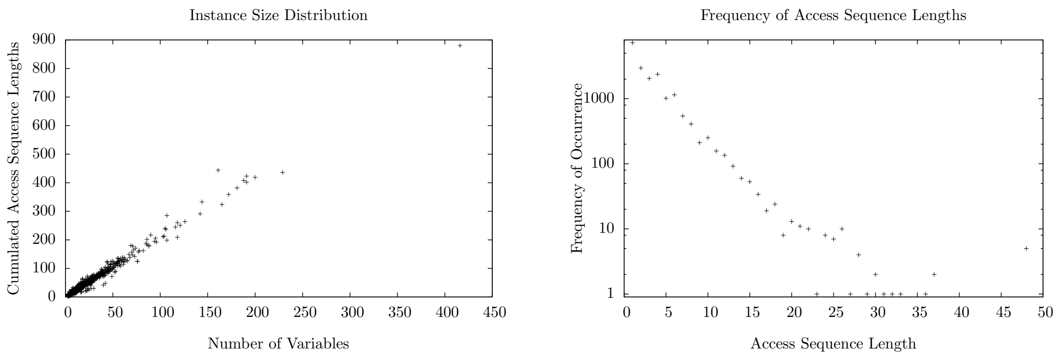


Figure 5.7: Distribution of the number of variables and access lengths across 1, 918 selected OffsetStone instances.

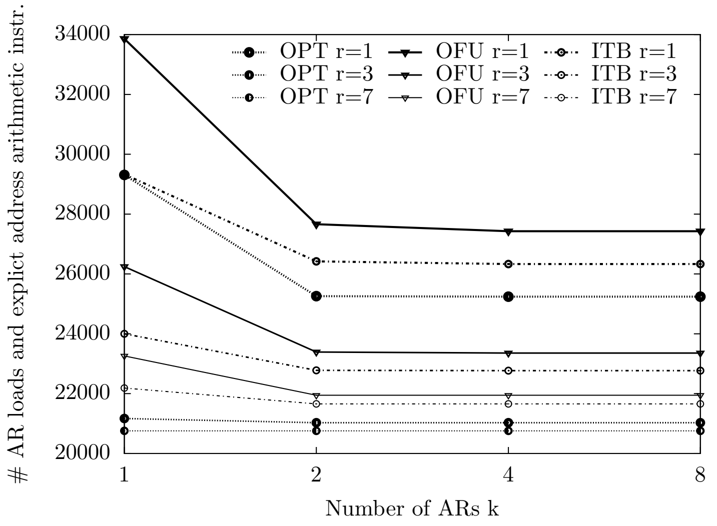


Figure 5.8: Offset Assignment Costs with varying autoin-/decrement ranges and numbers of address registers on 1, 918 selected OffsetStone instances.

Fig. 5.8 shows the impact of the various configurations for r and k on the total offset assignment cost, accumulated over all of the 1, 918 instances. The central observation is that the amount of address code can be considerably reduced when exploiting larger auto-modify ranges. At least for the evaluated instances, it appears that the reduction potential by increasing r is much higher than by increasing k since the offset assignment costs do not further decrease significantly for $k > 2$. However, this is partially also due to the already discussed character of the instances. The results approve the already in [ML14] observed impression, that the performance loss when using the proposed heuristics is rather small - also for increasing auto-modify ranges. Like for the SOA case, the `GOA-ITB-MCF`-layout is already a considerably better basis for the ARA part than an order-of-first-use layout. However, there is even more potential in achieving near-optimal solutions by generating memory layouts that are not SOA-oriented but already take larger auto-modify ranges into account. Using an optimum address register assignment is worthwhile and computationally not too intensive so that it can be performed in production compilers. Irrespective of the r-k combinations, the heuristics are fast; their running times sum up to less than half of a second, although many small min-cost flow problems need to be solved.

Bibliography

[ABCC06] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, Princeton, NJ, USA, 2006.

[ABZ88] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[AEBS08] H. S. Ali, H. M. El-Boghdadi, and S. I. Shaheen. A new heuristic for SOA problem based on effective tie break function. In *Proc. of the 11th Intern. Workshop on Softw. and Compilers for Embed. Syst.*, SCOPES '08, pages 53–59, New York, NY, USA, 2008. ACM.

[AGG98] E. R. Altman, R. Govindarajan, and G. R. Gao. A unified framework for instruction scheduling and mapping for function units with structural hazards. *J. Parallel Distrib. Comput.*, 49(2):259–293, Mar. 1998.

[ALSU86] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[AM09] C. Ambühl and M. Mastrolilli. Single machine precedence constrained scheduling is a vertex cover problem. *Algorithmica*, 53(4):488–503, 2009.

[ARK00] S. Atri, J. Ramanujam, and M. T. Kandemir. Improving offset assignment on embedded processors using transformations. In *Proc. of the 7th Intern. Conf. on High Perf. Comput.*, HiPC '00, pages 367–374. Springer, 2000.

[ARK01] S. Atri, J. Ramanujam, and M. T. Kandemir. Improving offset assignment for embedded processors. In *Proc. of the 13th Intern. Workshop on Lang. and Compilers for Parallel Comput.*, LCPC '00, pages 158–172. Springer, 2001.

[Ary85] S. Arya. An optimal instruction-scheduling model for a class of vector processors. *IEEE Trans. on Computers*, 34(11):981–995, 1985.

[Bal85] E. Balas. On the facial structure of scheduling polyhedra. In R. W. Cottle, editor, *Mathematical Programming Essays in Honor of George B. Dantzig Part I*, volume 24 of *Mathematical Programming Studies*, pages 179–218. Springer, 1985.

[Bar92] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exper.*, 22(2):101–110, 1992.

[BDM12] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems.* SIAM, Philadelphia, PA, USA, 2012.

[BEP$^+$07] J. Błażewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. *Handbook on Scheduling.* Springer, 2007.

[BG89] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Trans. Program. Lang. Syst.*, 11:57–66, Jan. 1989.

[BGJ$^+$14] J. Békési, G. Galambos, M. N. Jung, M. Oswald, and G. Reinelt. A branch-and-bound algorithm for the coupled task problem. *Mathematical Methods of Operations Research*, 80(1):47–81, 2014.

[BL99] S. Bashford and R. Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems*, 4(2-3):119–165, 1999.

[BLV95] E. Balas, J. K. Lenstra, and A. Vazacopoulos. The one-machine problem with delayed precedence constraints and its use in job shop scheduling. *Management Science*, 41(1):94–109, 1995.

[BNQW11] C. Bessiere, N. Narodytska, C.-G. Quimper, and T. Walsh. The alldifferent constraint with precedences. In T. Achterberg and J. C. Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6697 of *LNCS*, pages 36–52. Springer, 2011.

[Boe82] K. Boenchendorf. Reihenfolgeprobleme - mean-flow-time sequencing. In *Mathematical Systems in Economics*, volume 74. Verlagsgruppe Athenäum, Hain, Scriptor, Hanstein, 1982.

[Bow72] V. Bowman. Permutation polyhedra. *SIAM J. on Applied Mathematics*, 22(4):580–589, 1972.

[BRG89] D. Bernstein, M. Rodeh, and I. Gertner. Approximation algorithms for scheduling arithmetic expressions on pipelined machines. *J. Algorithms*, 10:120–139, Mar. 1989.

[Car82] J. Carlier. The one-machine sequencing problem. *European J. of Operational Research*, 11(1):42–47, 1982.

[CC95] H.-C. Chou and C.-P. Chung. An optimal instruction scheduler for superscalar processor. *IEEE Trans. on Parallel and Distributed Systems*, 6(3):303–313, 1995.

[CCK97] C.-M. Chang, C.-M. Chen, and C.-T. King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications*, 34(9):1–14, 1997.

[CCPS98] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[CF96] A. Caprara and M. Fischetti. $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts. *Mathematical Programming*, 74(3):221–235, 1996.

[CG72] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[CH99] F. A. Chudak and D. S. Hochbaum. A half-integral linear programming relaxation for scheduling precedence-constrained jobs on a single machine. *Operation Reseach Letters*, 25(5):199–204, Dec. 1999.

[Chv73] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4(4):305–337, 1973.

[CK02] Y. Choi and T. Kim. Address assignment combined with scheduling in DSP code generation. In *Proc. of the 39th Design Automation Conf.*, DAC '02, pages 225–230, New York, NY, USA, 2002. ACM.

[CM99] C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Appl. Math.*, 98(1-2):29–38, Nov. 1999.

[CPL13] CPLEX optimization studio version 12.6. Reference manual, IBM ILOG, 2013.

[CRdS06] P. E. Coll, C. C. Ribeiro, and C. C. de Souza. Multiprocessor scheduling under precedence constraints: Polyhedral results. *Discrete Applied Mathematics*, 154(5):770–801, 2006.

[CS05] J. R. Correa and A. S. Schulz. Single-machine scheduling with precedence constraints. *Mathematics of Operations Research*, 30(4):1005–1021, 2005.

[CSS98] K. D. Cooper, P. J. Schielke, and D. Subramanian. An experimental evaluation of list scheduling. Technical Report 98-326, Department of Computer Science, Rice University, Houston, TX, USA, 1998.

[DFJ54] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 3:393–410, 1954.

[DJK11] B. Dezső, A. Jüttner, and P. Kovács. LEMON - an open source C++ graph template library. *Electron. Notes in Theor. Comp. Sc.*, 264(5):23–45, 2011.

[DPL93] S. Dauzère-Pérès and J. B. Lasserre. A modified shifting bottleneck procedure for job-shop scheduling. *Intern. J. of Production Research*, 31(4):923–932, 1993.

[DW90] M. E. Dyer and L. A. Wolsey. Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics*, 26(2-3):255–270, 1990.

[Edm65] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *J. of Research of the National Bureau of Standards*, 69B(1-2):125–130, 1965.

[EGJR01] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi. Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In *Comput. Comb. Opt., Optimal or Provably Near-Optimal Solutions*, volume 2241 of *LNCS*, pages 157–222. Springer, 2001.

[Eis99] F. Eisenbrand. On the membership problem for the elementary closure of a polyhedron. *Combinatorica*, 19(2):297–300, 1999.

[EK91] M. A. Ertl and A. Krall. Optimal instruction scheduling using constraint logic programming. In J. Maluszyński and M. Wirsing, editors, *Programming Lang. Implem. and Logic Programming*, volume 528 of *LNCS*, pages 75–86. Springer, 1991.

[Eri11] M. Eriksson. *Integrated Code Generation.* PhD thesis, Linköping University, Dept. of Computer and Inform. Science, The Institute of Technology, 2011.

[Fio01] S. Fiorini. *Polyhedral Combinatorics of Order Polytopes.* PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, 2001.

[FL96] L. Finta and Z. Liu. Single machine scheduling subject to precedence delays. *Discrete Applied Mathematics*, 70(3):247–266, 1996.

[FL07] M. Fischetti and A. Lodi. Optimizing over the first Chvátal closure. *Mathematical Programming*, 110(1):3–20, 2007.

[GE91] C. H. Gebotys and M. I. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *Proc. of the 28th ACM/IEEE Design Automation Conf.*, DAC '91, pages 2–7, New York, NY, USA, 1991. ACM.

[Geb97] C. H. Gebotys. DSP address optimization using a minimum cost circulation technique. In *Proc. 1997 IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD '97, pages 100–103, 1997.

[Geb99] C. H. Gebotys. A minimum-cost circulation approach to DSP address-code generation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):726–741, 1999.

[GG77] P. Gaiha and S. Gupta. Adjacent vertices on a permutohedron. *SIAM J. on Applied Mathematics*, 32(2):323–327, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[GJ00] E. Gawrilow and M. Joswig. polymake: a framework for analyzing convex polytopes. In G. Kalai and G. M. Ziegler, editors, *Polytopes — Combinatorics and Computation*, pages 43–74. Birkhäuser, 2000.

[GJR84] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32(6):1195–1220, 1984.

[GJR85a] M. Grötschel, M. Jünger, and G. Reinelt. Facets of the linear ordering polytope. *Mathematical Programming*, 33(1):43–60, 1985.

[GJR85b] M. Grötschel, M. Jünger, and G. Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 33(1):28–42, 1985.

[GL81] P. Gács and L. Lovász. Khachiyan's algorithm for linear programming. In H. König, B. Korte, and K. Ritter, editors, *Mathematical Programming at Oberwolfach*, volume 14 of *Mathematical Programming Studies*, pages 61–68. Springer, 1981.

[GLLK79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Discrete Optimization II, Proc. of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symp.*, volume 5 of *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979.

[GLS81] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[GLS84] M. Grötschel, L. Lovász, and A. Schrijver. Corrigendum to our paper "the ellipsoid method and its consequences in combinatorial optimization". *Combinatorica*, 4(4):291–295, 1984.

[Gom58] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

[Gom63] R. E. Gomory. An algorithm for integer solutions to linear programs. In R. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, New York, USA, 1963.

[GP79a] M. Grötschel and M. W. Padberg. On the symmetric travelling salesman problem I: Inequalities. *Mathematical Programming*, 16(1):265–280, 1979.

[GP79b] M. Grötschel and M. W. Padberg. On the symmetric travelling salesman problem II: Lifting theorems and facets. *Mathematical Programming*, 16(1):282–302, 1979.

[GP85] M. Grötschel and M. W. Padberg. Polyhedral theory. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem*, chapter 8, pages 251–305. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[GR63] G. T. Guilbaud and P. Rosenstiehl. Analyse algébrique d'un scrutin. *Mathématiques et Sciences Humaines*, 4:9–33, 1963.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. of Applied Mathematics*, 17(2):416–429, 1969.

[HABT11] J. Huynh, J. N. Amaral, P. Berube, and S.-A.-A. Touati. Evaluating address register assignment and offset assignment algorithms. *ACM Trans. Embed. Comput. Syst.*, 10(3):37:1–37:22, 2011.

[Hal35] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10(1):26–30, 1935.

[HB01] S. Haga and R. Barua. EPIC instruction scheduling based on optimal approaches. In *In Proc. 1st Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, pages 22–31, 2001.

[HG83] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, 5:422–448, July 1983.

[HK56] A. J. Hoffman and J. B. Kruskal. Integral boundary points of convex polyhedra. In H. W. Kuhn and A. W. Tucker, editors, *Linear Inequalities and Related Systems*, volume 38 of *Annals of Mathematics Studies*, pages 223–246. Princeton University Press, Princeton, NJ, USA, 1956.

[Hon72] S. Hong. *A linear programming approach for the traveling salesman problem.* PhD thesis, The Johns Hopkins University, Baltimore, USA, 1972.

[Hoo00] J. N. Hooker. *Continuous Relaxations*, pages 225–270. John Wiley & Sons, Inc., 2000.

[Hoo11] J. N. Hooker. Hybrid modeling. In P. van Hentenryck and M. Milano, editors, *Hybrid Optimization*, volume 45 of *Springer Optimization and Its Applications*, pages 11–62. Springer, 2011.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2007.

[HSSW97] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22(3):513–544, 1997.

[HW05] M. Heffernan and K. Wilken. Data-dependency graph transformations for instruction scheduling. *J. of Scheduling*, 8(5):427–451, Oct. 2005.

[Iba76] T. Ibaraki. Integer programming formulation of combinatorial optimization problems. *Discrete Mathematics*, 16(1):39–52, 1976.

[JM13] M. Jünger and S. Mallach. Solving the simple offset assignment problem as a traveling salesman. In *Proc. 16th Int. W. on Softw. and Compilers for Embed. Syst.*, M-SCOPES '13, pages 31–39. ACM, 2013.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proc. of the 16th Annual ACM Symp. on Theory of Computing*, STOC '84, pages 302–311, New York, NY, USA, 1984. ACM.

[Käs00] D. Kästner. *Retargetable Postpass Optimisation by Integer Linear Programming*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[KB06] C. Kessler and A. Bednarski. Optimal integrated VLIW code generation with integer linear programming. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *Euro-Par 2006 Parallel Processing*, volume 4128 of *LNCS*, pages 461–472. Springer, 2006.

[Kha79] L. G. Khachiyan. A polynomial algorithm in linear programming (in russian). *Doklady Akademii Nauk SSSR*, 20:191–194, 1979.

[KK12] Z. Király and P. Kovács. Efficient implementations of minimum-cost flow algorithms. *Acta Universitatis Sapientiae, Informatica*, 4(1):67–118, 2012.

[KKF09] A. B. Keha, K. Khowala, and J. W. Fowler. Mixed integer programming formulations for single machine scheduling problems. *Comput. Ind. Eng.*, 56(1):357–367, Feb. 2009.

[KP80] R. M. Karp and C. H. Papadimitriou. On linear characterizations of combinatorial optimization problems. In *21st Annual Symposium on Foundations of Computer Science*, pages 1–9, Oct. 1980.

[KP82] R. M. Karp and C. H. Papadimitriou. On linear characterizations of combinatorial optimization problems. *SIAM Journal on Computing*, 11:620–632, 1982.

[KV12] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics.* Springer, fifth edition, 2012.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[LBM98] R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in DSP programs. In *Proc. of the ASP-DAC '98: Asia and South Pacific Design Automation Conf. 1998*, pages 87–92, Feb. 1998.

[LC96] M. Langevin and E. Cerny. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. Des. Autom. Electron. Syst.*, 1(4):443–455, Oct. 1996.

[LCBS14] R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. Combinatorial spill code optimization and ultimate coalescing. *SIGPLAN Not.*, 49(5):23–32, June 2014.

[LD98] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proc. of the 11th Intern. Symp. on Syst. Synth.*, ISSS '98, pages 3–8, Washington, DC, USA, 1998. IEEE Computer Society.

[Leu03] R. Leupers. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *Proc. of the 12th Intern. Conf. on Compiler Constr.*, CC'03, pages 290–302. Springer, 2003.

[Lia96] S. Liao. *Code generation and optimization for embedded digital signal processors.* PhD thesis, Massachusetts Institute of Technology, 1996.

[LKB$^+$01] M. Lorenz, D. Kottmann, S. Bashford, R. Leupers, and P. Marwedel. Optimized address assignment for DSPs with SIMD memory accesses. In *Proc. 2001 Asia and South Pacific Design Autom. Conf.*, ASP-DAC '01, pages 415–420. ACM, 2001.

[LM96] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proc. of the 1996 IEEE/ACM Intern. Conf. on Computer-Aided Design*, ICCAD '96, pages 109–112, Washington, DC, USA, 1996. IEEE Computer Society.

[LM97] R. Leupers and P. Marwedel. Time-constrained code compaction for DSP's. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(1):112–122, Mar. 1997.

[LM10] P. Lokuciejewski and P. Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems.* Springer, Nov. 2010.

[LOQTvB03] A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proc. of the 18th Intern. Joint Conf. on Artificial Intelligence*, IJCAI'03, pages 245–250, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers, Inc.

[LS14] R. Castañeda Lozano and C. Schulte. Survey on combinatorial register allocation and instruction scheduling. *CoRR*, abs/1409.7628, 2014.

[Mal08] A. M. Malik. *Constraint Programming Techniques for Optimal Instruction Scheduling.* PhD thesis, University of Waterloo, Waterloo, Canada, 2008.

[Mal14] S. Mallach. More general optimal offset assignment. Paper preprint, Institut für Informatik, Universität zu Köln, Cologne, Germany, 2014.

[Mar06] P. Marwedel. *Embedded System Design.* Springer, 2006.

[McC76] G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I - convex underestimating problems. *Mathematical Programming*, 10(1):147–175, 1976.

[Min96] H. Minkowski. *Geometrie der Zahlen (Erste Lieferung).* Teubner, Leipzig, Germany, 1896. reprinted: Chelsea, New York, 1953.

[ML14] S. Mallach and R. Castañeda Lozano. Optimal general offset assignment. In *Proc. 17th Int. W. on Softw. and Compilers for Embed. Syst.*, SCOPES '14, pages 50–59, New York, NY, USA, 2014. ACM.

[MMvB06] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraing programming. In *Proc. 18th IEEE Intern. Conf. on Tools with Artificial Intelligence (ICTAI '06)*, pages 279–287, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[MMvB08] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraing programming. *Int. J. on Artificial Intelligence Tools*, 17(1):37–54, 2008.

[MQW03] F. Margot, M. Queyranne, and Y. Wang. Decompositions, network flows, and a precedence constrained single-machine scheduling problem. *Operations Research*, 51(6):981–992, Nov. 2003.

[MR11] R. Martí and G. Reinelt. *The Linear Ordering Problem.* Springer, 2011.

[MT00] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In R. Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000*, volume 1894 of *LNCS*, pages 306–319. Springer, 2000.

[NS92] G. L. Nemhauser and M. W. P Savelsbergh. A cutting plane algorithm for the single machine scheduling problem with release times. In M. Akgül, H. W. Hamacher, and S. Tüfekçi, editors, *Combinatorial Optimization*, volume 82 of *NATO ASI Series*, pages 63–83. Springer, 1992.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., New York, NY, USA, 1988.

[OKT06] O. Ozturk, M. T. Kandemir, and S. Tosun. An ILP based approach to address code generation for digital signal processors. In G. Qu, Y. I. Ismail, N. Vijaykrishnan, and H. Zhou, editors, *ACM Great Lakes Symp. on VLSI*, pages 37–42, Philadelphia, PA, USA, Apr. 2006. ACM.

[OOAL06] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers. Offset assignment using simultaneous variable coalescing. *ACM Trans. Embed. Comput. Syst.*, 5(4):864–883, Nov. 2006.

[Orl96] J. B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. In *Proc. of the 7th Annual ACM-SIAM Symp. on Discrete Algorithms*, SODA '96, pages 474–481, Philadelphia, PA, USA, 1996. SIAM.

[Pet88] R. Peters. L'ordonnancement sur une machine avec des contraintes de délai. *Belgian J. of Operations Research, Statistics and Computer Science*, 28:33–76, 1988.

[PJ00] H. P. Peixoto and M. F. Jacome. A new technique for estimating lower bounds on latency for high level synthesis. In *Proc. of the 10th Great Lakes Symp. on VLSI*, GLSVLSI '00, pages 129–132, New York, NY, USA, 2000. ACM.

[Pot80] C. N. Potts. An algorithm for the single machine sequencing problem with precedence constraints. In V. J. Rayward-Smith, editor, *Combinatorial Optimization II*, volume 13 of *Mathematical Programming Studies*, pages 78–87. Springer, 1980.

[PR81] M. W. Padberg and M. R. Rao. The russian method for linear inequalities, part III. Research Report 81-39, New York University, 1981.

[PR82] M. W. Padberg and M. R. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7(1):67–80, 1982.

[PR90] M. W. Padberg and G. Rinaldi. Facet identification for the symmetric traveling salesman polytope. *Math. Program.*, 47(2):219–257, June 1990.

[PS93] K. V. Palem and B. B. Simons. Scheduling time-critical instructions on RISC machines. *ACM Trans. Program. Lang. Syst.*, 15:632–658, Sep. 1993.

[Pug98] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proc. of the 15th National/10th Conf. on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 359–366, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[QS94] M. Queyranne and A. S. Schulz. Polyhedral approaches to machine scheduling. Technical Report 408/1994, Technische Universität Berlin, Berlin, Germany, 1994. Revised in 1996.

[Que93] M. Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*, 58(1-3):263–285, 1993.

[QW91a] M. Queyranne and Y. Wang. A cutting plane procedure for precedence-constrained single machine scheduling. Working paper, Faculty of Commerce, University of British Columbia, Vancouver, B.C, 1991.

[QW91b] M. Queyranne and Y. Wang. Single-machine scheduling polyhedra with precedence constraints. *Mathematics of Operations Research*, 16(1):1–20, 1991.

[RJ94] M. Rim and R. Jain. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 13:451–458, Apr. 1994.

[RP99] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. *SIGPLAN Not.*, 34(5):128–138, May 1999.

[Sch11] P. H. Schoute. Analytic treatment of the polytopes regularly derived from the regular polytopes. In *Verhandelingen der Koninklijke Akademie van Wetenschappen te Amsterdam (Eerste Sectie)*, volume 11. Johannes Müller, Amsterdam, The Netherlands, 1911.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Sch96] A. S. Schulz. Scheduling to minimize total weighted completion time: Performance guarantees of LP-based heuristics and lower bounds. In W. H. Cunningham, S. T. McCormick, and M. Queyranne, editors, *Integer Programming and Combinatorial Optimization*, volume 1084 of *LNCS*, pages 301–315. Springer, 1996.

[SEB12] H. Shokry and H. M. El-Boghdadi. On heuristic solutions to the simple offset assignment problem in address-code optimization. *ACM Trans. Embed. Comput. Syst.*, 11(3):63:1–63:17, Sep. 2012.

[SINF96] N. Sugino, S. Iimuro, A. Nishihara, and N. Fujii. DSP code optimization utilizing memory addressing operation. *IEICE Trans. on Fundamentals of Electronics, Communication and Computer Sciences*, 79(8):1217–1224, 1996.

[Sou89] J. P. Sousa. *Time Indexed Formulations of Non-Preemptive Single-Machine Scheduling Problems.* PhD thesis, Université Catholique de Louvain, Louvain, Belgium, 1989.

[SR07] H. Salamy and J. Ramanujam. An effective heuristic for simple offset assignment with variable coalescing. In *Proc. of the 19th Intern. Conf. on Lang. and Compilers for Parallel Comput.*, LCPC'06, pages 158–172. Springer, 2007.

[SR12] H. Salamy and J. Ramanujam. An ILP solution to address code generation for embedded applications on digital signal processors. *ACM Trans. Des. Autom. Electron. Syst.*, 17(3):28:1–28:23, June 2012.

[Sve11] O. Svensson. Hardness of precedence constrained scheduling on identical machines. *SIAM Journal on Computing*, 40(5):1258–1274, 2011.

[Tar85] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

[TC98] G. Tiruvuri and M. Chung. Estimation of lower bounds in scheduling algorithms for high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 3(2):162–180, Apr. 1998.

[Tep13] L. Tepaße. *Untersuchung einer IP-Formulierung und Entwicklung eines Branch & Cut-Algorithmus zur Lösung des Single-Issue Instruction Scheduling Problems.* Master thesis, Universität zu Köln, Cologne, Germany, 2013.

[Thi95] S. Thienel. *ABACUS - A Branch-And-CUt System.* PhD thesis, Universität zu Köln, Cologne, Germany, 1995.

[vAFS90] A. von Arnim, U. Faigle, and R. Schrader. The permutahedron of series-parallel posets. *Discrete Applied Mathematics*, 28(1):3–9, 1990.

[vBW01] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proc. of the 7th Intern. Conf. on Principles and Practice of Constraint Programming*, CP '01, pages 625–639. Springer, 2001.

[Veg92] S. R. Vegdahl. A dynamic-programming technique for compacting loops. *SIGMICRO Newsl.*, 23(1-2):180–188, Dec. 1992.

[vH01] W. J. van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001.

[Vil07] P. Vilím. *Global Constraints in Scheduling.* PhD thesis, Charles University, Prague, Czech Republic, Aug. 2007.

[Vil11] P. Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In T. Achterberg and J. Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6697 of *LNCS*, pages 230–245. Springer, 2011.

[Wey35] H. Weyl. Elementare Theorie der konvexen Polyeder. *Commentarii Mathematici Helvetici*, 7:290–306, 1935.

[WG97] B. Wess and M. Gotschlich. Optimal DSP memory layout generation as a quadratic assignment problem. In *Proc. 1997 IEEE Intern. Symp. on Circuits and Systems ISCAS '97*, volume 3, pages 1712–1715, June 1997.

[WLH00] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. *SIGPLAN Not.*, 35:121–133, May 2000.

[WMGB93] T. C. Wilson, N. Mukherjee, M. K. Garg, and D. K. Banerji. An integrated and accelerated ILP solution for scheduling, module allocation, and binding in datapath synthesis. In *Proc. of 6th Int. Conf. on VLSI Design*, pages 192–197, 1993.

[Wol90] L. A. Wolsey. Formulating single machine scheduling problems with precedence constraints. In J. J. Gabszewic, J. F. Richard, and L. A. Wolsey, editors, *Economic Decision-Making: Games, Econometrics and Optimisation*, pages 473–484. North-Holland, 1990.

[Wol98] L. A. Wolsey. *Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[WY01] H. P. Williams and H. Yan. Representations of the all_different predicate of constraint satisfaction in integer programming. *INFORMS J. on Computing*, 13(2):96–103, 2001.

[You78] H. P. Young. On permutations and permutation polytopes. In M. L. Balinski and A. J. Hoffman, editors, *Polyhedral Combinatorics*, volume 8 of *Mathematical Programming Studies*, pages 128–140. Springer, 1978.

[Zha96] L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1996.

[Zie95] G. M. Ziegler. *Lectures on polytopes*. Springer, New York, 1995.

Erklärung

Ich versichere, dass ich die von mir vorgelegte Dissertation selbstständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie - abgesehen von unten angegebenen Teilpublikationen - noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen der Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr. Michael Jünger betreut worden.

Nachfolgend genannte Teilpublikationen liegen vor:

M. Jünger and S. Mallach. *An Integer Programming Approach to Optimal Basic Block Instruction Scheduling for Single-Issue Processors.* Paper preprint, Institut für Informatik, Universität zu Köln, Cologne, Germany, 2014.

M. Jünger and S. Mallach. *Solving the Simple Offset Assignment Problem as a Traveling Salesman.* In *Proc. 16th Int. W. on Softw. and Compilers for Embed. Syst., M-SCOPES '13*, pages 31-39. ACM, 2013.

S. Mallach and R. Castañeda Lozano. *Optimal General Offset Assignment.* In *Proc. 17th Int. W. on Softw. and Compilers for Embed. Syst., SCOPES '14*, pages 50-59, ACM, 2014.

S. Mallach. *More general optimal offset assignment.* Paper preprint, Institut für Informatik, Universität zu Köln, Cologne, Germany, 2014.

Ich versichere, dass ich alle Angaben wahrheitsgemäß nach bestem Wissen und Gewissen gemacht habe und verpflichte mich, jedmögliche, die obigen Angaben betreffenden Veränderungen, dem Dekanat unverzüglich mitzuteilen.

Köln, 10. März 2015

S. Mallach